Handbook of Magma Functions

Volume 5

Finite Groups

John Cannon Wieb Bosma
Claus Fieker Allan Steel

Editors

Version 2.19

Sydney

April 24, 2013

MAGMA
COMPUTER ALGEBRA

Handbook of Magma Functions

Editors:

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

Handbook Contributors:

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

Production Editors:

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

HTML Production:

Claus Fieker *Allan Steel*

VOLUME 5: OVERVIEW

VOLUME 5: CONTENTS

PART IX
FINITE GROUPS

57 GROUPS

Chapter 57
GROUPS

57.1 Introduction

Groups arise in several different categories in MAGMA. In the case of the category of permutation groups and the category of soluble groups defined by a power-conjugate presentation, all groups in the category are finite. However, the finitely-presented group category, the polycyclic group category, the abelian group category and the matrix group category contain both finite and infinite groups. In the case of the abelian group category and the matrix group category, a large number of functions are available for finite groups only. In the near future, these functions will be extended to finite finitely-presented groups of moderate order.

In this chapter, we discuss the functions that are provided for groups collectively, noting especially those functions that are available only for finite groups. Descriptions of functions that depend upon the particular category may be found in the chapter devoted to that category.

57.1.1 The Categories of Finite Groups

At present MAGMA contains five main categories of finite groups:

(i) Permutation groups: category `GrpPerm`;

(ii) Finite matrix groups: category `GrpMat`;

(iii) Finite solvable groups given by a power-conjugate presentation: category `GrpPC`;

(iv) Finite abelian groups: category `GrpAb`;

(v) Finite polycyclic groups: category `GrpGPC`.

Note that the categories `GrpMat`, `GrpAb` and `GrpGPC` contain both finite and infinite groups; most of the operations described in this chapter apply only to finite groups belonging to these categories. In this chapter we will use the category name `GrpFin` to collectively refer to categories `GrpPerm` and `GrpPC` and the subcategories of `GrpMat`, `GrpAb` and `GrpGPC` consisting of finite groups. The category name `Grp` will be used when the operation does not depend upon the finiteness of the group.

57.2 Construction of Elements

57.2.1 Construction of an Element

Throughout this subsection we shall assume that the carrier set for the group G is a subset of the set S. Thus, if G is a permutation group on the set X, its carrier set will be a subset of $\mathrm{Sym}(X)$.

`elt< G | L >`

Given a group G whose elements are a subset of the set S, and a list L of objects $a_1, a_2, \ldots, a_n$ defining an element of S, construct this element g of S. Then, the element g will be tested for membership of G, and if g is not an element of G, the function will fail. If g does lie in G, g will be returned with G as its parent.

`G ! Q`

Given a group G whose elements are a subset of the set S, and a sequence $Q = [a_1, a_2, \ldots, a_n]$ defining an element of S, construct this element g of S. Then, the element g will be tested for membership of G, and if g is not an element of G, the function will fail. If g does lie in G, g will be returned with G as its parent.

`Identity(G)`

`Id(G)`

Construct the identity element in the group G.

57.2.2 Coercion

`G ! g`

Given a group G and an element g of H, where G and H are subgroups of some common over-group and g is contained in G, embed g in G. Thus this operator changes the parent of g into G. The coercion may fail for groups in the category `GrpFP`.

57.2.3 Homomorphisms

`hom< G -> H | L >`

Return the group homomorphism $\phi : G \to H$ defined by extending the map of the generators of G, as given by the list L on the right side of the constructor. Suppose that the generators of G are $g_1, \ldots, g_n$, and that $\phi(g_i) = h_i$ for each i. Then L must be one of the following:

(a) a list of the n 2-tuples $< g_i, h_i >$ (order not important);

(b) a list of the n arrow-pairs g_i `->` h_i (order not important);

(c) $h_1, \ldots, h_n$ (order is important).

For its computations, MAGMA often assumes that the mapping so defined is a homomorphism without attempting to verify this.

For certain categories of groups, e.g. `GrpGPC`, the homomorphism constructor provides some additional functionality. See the chapter on the appropriate category for further information.

`hom< G -> H | x :-> e(x) >`

Return the group homomorphism $\phi : G \to H$ defined by the rule $\phi(x) = e(x)$, where x is a general element of G and $e(x)$ is an expression in x. The symbol x may be any identifier name, and has local scope. For its computations, MAGMA assumes the expression defines a homomorphism, but does not verify this.

`IdentityHomomorphism(G)`

Return the identity homomorphism $\phi : G \to G : x \mapsto x$.

Example H57E1

Construction of an isomorphism from the cyclic group of order 15 to the abelian group isomorphic to $\mathbf{Z}/15\mathbf{Z}$, by giving the image of the generator:

```
> C15 := CyclicGroup(15);
> C15;
Permutation group C15 acting on a set of cardinality 15
Order = 15 = 3 * 5
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
> A15 := AbelianGroup([15]);
> A15;
Abelian Group isomorphic to Z/15
Defined on 1 generator
Relations:
    15*A15.1 = 0
> iso11 := hom< C15 -> A15 | C15.1 -> 11*A15.1 >;
> A15 eq iso11(C15);
true
> forall{ <c, d> : c, d in C15 | iso11(c * d) eq iso11(c) * iso11(d) };
true
```

Example H57E2

An endomorphism of the same cyclic group, defined using an expression. The image is cyclic of order 5.

```
> C15 := CyclicGroup(15);
> h := hom< C15 -> C15 | g :-> g^3 >;
> forall{ <c, d> : c, d in C15 | h(c * d) eq h(c) * h(d) };
true
> im := h(C15);
> im;
Permutation group im acting on a set of cardinality 15
Order = 5
```

```
    (1, 4, 7, 10, 13)(2, 5, 8, 11, 14)(3, 6, 9, 12, 15)
> IsCyclic(im);
true
```

57.2.4 Arithmetic with Elements

`g * h`

Product of element g and element h, where g and h belong to the same generic group U. If g and h both belong to the same proper subgroup G of U, then the result will be returned as an element of G; if g and h belong to subgroups H and K of a subgroup G of U, then the product is returned as an element of G. Otherwise, the product is returned as an element of U. The product in abelian groups is called the sum and is written `g + h` instead.

`g ^ n`

The n-th power of the group element g, where n is a positive, negative or zero integer. In abelian groups, this is written as a scalar product `n * g` instead.

`g / h`

Product of the group element g by the inverse of the group element h, i.e., the element gh^{-1}. Here g and h must belong to the same generic group U. The rules for determining the parent group of g/h are the same as for gh. In abelian groups, this is written additively as `g - h`.

`g ^ h`

Conjugate of the group element g by the group element h, i.e., the element $h^{-1}gh$. Here g and h must belong to the same generic group U. The rules for determining the parent group of g^h are the same as for gh. In abelian groups, this operation does not exist.

`(g, h)`

Commutator of the group elements g and h, i.e., the element $g^{-1}h^{-1}gh$. Here g and h must belong to the same generic group U. The rules for determining the parent group of (g, h) are the same as those for gh.

`(g1, ..., gr)`

Given r elements $g_1, \ldots, g_r$ belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

`g eq h`

Given elements g and h belonging to the same generic group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given elements g and h belonging to the same generic group, return `true` if g and h are distinct elements, `false` otherwise.

`IsId(g)`

`IsIdentity(g)`

Returns `true` if the group element g is the identity element.

`Order(g)`

The order of the group element g.

Example H57E3

We illustrate the arithmetic operations by applying them to some elements of Sym(9).

```
> G := Sym(9);
> x := G ! (1,2,4)(5,6,8)(3,9,7);
> y := G ! (4,5,6)(7,9,8);
> x*y;
(1, 2, 5, 4)(3, 8, 6, 7)
> x^-1;
(1, 4, 2)(3, 7, 9)(5, 8, 6)
> x^2;
(1, 4, 2)(3, 7, 9)(5, 8, 6)
> x / y;
(1, 2, 6, 9, 8, 4)(3, 7)
> x^y;
(1, 2, 5)(3, 8, 9)(4, 7, 6)
> (x, y);
(1, 7, 3, 6)(4, 5, 9, 8)
> x^y eq y^x;
false
> CycleStructure(x^2*y);
[ <6, 1>, <2, 1>, <1, 1> ]
> Degree(y);
6
> Order(x^2*y);
6
```

57.3 Construction of a General Group

57.3.1 The General Group Constructors

The chapters on the individual group categories describe several methods for constructing groups; this section indicates one approach only.

```
PermutationGroup< X | L >
```

```
PermutationGroup< n | L >
```

```
MatrixGroup< n, R | L >
```

These expressions construct, respectively: a permutation group G acting on the set X; a permutation group G acting on the set $X = \{1, \ldots, n\}$; or a matrix group G of degree n over the ring R. The generic group U of which G is a subgroup will be $\mathrm{Sym}(X)$ in the permutation case or $\mathrm{GL}(n, R)$ in the matrix case. There are two return values: G, and the inclusion homomorphism from G to U.

The generators of G are defined by the list L. Each term of L must be an object of one of the following types:

(a) Either (permutation case) a sequence of n elements of X, or (matrix case) a sequence of n^2 elements of R, defining an element of U;

(b) A set or sequence of sequences of type (a);

(c) An element of U;

(d) A set or sequence of elements of U;

(e) A subgroup of U;

(f) A set or sequence of subgroups of U.

Each element or group specified by the list must belong to the *same* generic group. The group G will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of G consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list. Repetitions of an element and occurrences of the identity element are removed (unless G is trivial).

The `PermutationGroup` constructor is shorthand for the two statements:

```
U := SymmetricGroup(X);
G := sub< U | L >;
```

and the `MatrixGroup` constructor is shorthand for the two statements:

```
U := GeneralLinearGroup(n, R);
G := sub< U | L >;
```

where `sub< ... >` is the subgroup constructor described in the next subsection.

```
Group< X | R >
```

```
PolycyclicGroup< X | R >
```

```
AbelianGroup< X | R >
```

These expressions construct, respectively, a finitely presented group, a finite soluble group given by a power-conjugate presentation or a polycyclic group, and an abelian group, in the categories `GrpFP`, `GrpPC` or `GrpGPC`, and `GrpAb`. Given a list X of identifier names $x_1, \ldots, x_r$, and a list of relations R over them, first construct the free group F (in `GrpFP` or `GrpAb`) on the generators $x_1, \cdots, x_r$, and then construct the quotient G of F corresponding to the normal subgroup of F defined by the relations R. There are two return values: G, and the natural homomorphism from F to G.

The relations of G are defined by the list R. Each term of R must be an object of one of the following types:

(a) A word w of F, interpreted as the relator $w =$ identity of F;

(b) A relation $w_1 = w_2$, where w_1 and w_2 are words of F;

(c) A relation list $w_1 = w_2 = \cdots = w_r$, where the w_i are words of F, interpreted as the set of relations $w_1 = w_r, \ldots, w_{r-1} = w_r$.

Within R, the identity element of F may be represented by the digit `1` for `Group` or `PolycyclicGroup`, and `0` for `AbelianGroup`.

The construct $x_1, \ldots, x_n$ defines names for the generators of G that are local to the constructor, i.e., they are used when writing down the relations to the right of the bar. However, no assignment of *values* to these identifiers is made. If the user wants to refer to the generators by these (or other) names, then the *generators assignment* construct must be used on the left hand side of an assignment statement.

The constructor `PolycyclicGroup` returns either a finite soluble group given by a power-conjugate presentation (category `GrpPC`) or a general polycyclic group (category `GrpGPC`), depending on the arguments. R must be either a valid power-conjugate presentation for a finite soluble group or a consistent polycyclic presentation. If R is a valid power-conjugate presentation for a finite soluble group, a group in the category `GrpPC` is returned, unless the parameter `Class` is set to `"GrpGPC"`. If the parameter `Class` is set to `"GrpGPC"` or if R is not a valid power-conjugate presentation for a finite soluble group and the parameter `Class` is not set to `"GrpPC"`, a general polycyclic group in the category `GrpGPC` is returned. In any case, the free group F is in the category `GrpFP`. If R is neither a valid power-conjugate presentation for a finite soluble group nor a consistent polycyclic presentation, or if R does not match the value of the parameter `Class`, a runtime error is caused.

For a detailed description of this constructor and in particular for a description of power-conjugate presentations and consistent polycyclic presentations, we refer to Chapter 63 and Chapter 72, respectively.

Example H57E4

(1) The permutation group of degree 8 generated by the permutations $(1,7,2,8)(3,6,4,5)$ and $(1,4,2,3)(5,7,6,8)$:

```
> G := PermutationGroup< 8 |
>     (1, 7, 2, 8)(3, 6, 4, 5), (1, 4, 2, 3)(5, 7, 6, 8) >;
> G;
Permutation group G acting on a set of cardinality 8
    (1, 7, 2, 8)(3, 6, 4, 5)
    (1, 4, 2, 3)(5, 7, 6, 8)
```

(2) A matrix group of degree 2 over $\mathbf{F}_9$:

```
> K<w> := GF(9);
> M := MatrixGroup< 2, K | [w,w,1,2*w], [0,2*w,1,1], [1,0,1,2] >;
> M;
MatrixGroup(2, GF(3^2))
Generators:
    [  w   w]
    [  1 w^5]

    [  0 w^5]
    [  1   1]

    [  1   0]
    [  1   2]
> Order(M);
5760
```

(3) The finitely presented group Q defined by the presentation

$$< s,t,u \mid t^2, u^{17}, s^2 = t^s = t, u^s = u^{16}, u^t = u >,$$

together with the natural homomorphism from the free group to Q:

```
> Q<s,t,u>, h := Group< s, t, u |
>     t^2, u^17, s^2 = t^s = t, u^s = u^16, u^t = u >;
> Q;
Finitely presented group Q on 3 generators
Relations
    t^2 = Id(Q)
    u^17 = Id(Q)
    s^2 = t
    t^s = t
    u^s = u^16
    u^t = u
> Domain(h);
Finitely presented group on 3 generators (free)
```

(4) The soluble group of order 70 defined by the presentation $< a,b,c \mid a^2 = b, b^5 = c, c^7 >$:

```
> G<a,b,c> := PolycyclicGroup< a, b, c | a^2 = b, b^5 = c, c^7 >;
```

```
> G;
GrpPC : G of order 70 = 2 * 5 * 7
PC-Relations:
a^2 = b,
b^5 = c,
c^7 = Id(G)
```

(5) A finite abelian group on 4 generators:

```
> G := AbelianGroup< h, i, j, k | 5*h, 4*i, 7*j, 2*k - h >;
> G;
Abelian Group isomorphic to Z/2 + Z/140
Defined on 4 generators
Relations:
    G.1 + 8*G.4 = 0
    4*G.2 = 0
    7*G.3 = 0
    10*G.4 = 0
> Order(G);
280
```

Example H57E5

Using the constructor `PolycyclicGroup` with different values of the parameter `Class`, we construct the dihedral group of order 10 first as a finite soluble group given by a power-conjugate presentation (`GrpPC`) and next as a general polycyclic group (`GrpGPC`). Note that the presentation $\langle a, b \,|\, a^2, b^5, b^a = b^4 \rangle$ is both a valid power-conjugate presentation and a consistent polycyclic presentation, so we have to set the parameter `Class` to `"GrpGPC"` if we want to construct a group in the category `GrpGPC`.

```
> G1<a,b> := PolycyclicGroup< a,b | a^2, b^5, b^a=b^4 >;
> G1;
GrpPC : G1 of order 10 = 2 * 5
PC-Relations:
    a^2 = Id(G1),
    b^5 = Id(G1),
    b^a = b^4
> G2<a,b> := PolycyclicGroup< a,b | a^2, b^5, b^a=b^4 : Class := "GrpGPC">;
> G2;
GrpGPC : G2 of order 10 = 2 * 5 on 2 PC-generators
PC-Relations:
    a^2 = Id(G2),
    b^5 = Id(G2),
    b^a = b^4
```

We construct the infinite dihedral group as a group in the category `GrpGPC` from a consistent polycyclic presentation. We do not have to use the parameter `Class` in this case.

```
> G3<a,b> := PolycyclicGroup< a,b | a^2, b^a=b^-1>;
> G3;
```

```
GrpGPC : G3 of infinite order on 2 PC-generators
PC-Relations:
    a^2 = Id(G3),
    b^a = b^-1
```

The presentation $\langle a, b \mid a^2, b^4, b^a = b^3 \rangle$ is not a valid power-conjugate presentation for the dihedral group of order 8, since the exponent of b is not prime. However, it is a consistent polycyclic presentation. Consequently, the constructor `PolycyclicGroup` without specifying a value for the parameter `Class` returns a group in the category `GrpGPC`.

```
> G4<a,b> := PolycyclicGroup< a,b | a^2, b^4, b^a=b^3 >;
> G4;
GrpGPC : G4 of order 2^3 on 2 PC-generators
PC-Relations:
    a^2 = Id(G3),
    b^4 = Id(G3),
    b^a = b^3
```

57.3.2 Construction of Subgroups

`sub< G | L >`

Given the group G, construct the subgroup H of G, generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) A MAGMA object which may be coerced into G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic group. The subgroup H will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of H consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list. Repetitions of an element and occurrences of the identity element are removed (unless H is trivial).

`ncl< G | L >`

Given the group G, construct the subgroup H of G that is the *normal closure* of the subgroup H generated by the elements specified by the list L, where the possibilities for L are the same as for the `sub`-constructor.

Example H57E6

Let Q be the finitely presented group in generators s, t, u constructed in an earlier example. We construct the subgroup S of Q generated by ts^2 and u^4:

```
> Q<s,t,u>, h := Group< s, t, u |
>     t^2, u^17, s^2 = t^s = t, u^s = u^16, u^t = u >;
> S := sub< Q | t*s^2, u^4 >;
> S;
Finitely presented group S on 2 generators
Generators as words
    S.1 = $.2 * $.1^2
    S.2 = $.3^4
```

57.3.3 Construction of Quotient Groups

`quo< G | L >`

Given the group G, construct the quotient group $Q = G/N$, where N is the normal closure of the subgroup of G generated by the elements specified by L. The clause L is a list of one or more items of the following types:

(a) A MAGMA object which can be coerced into G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic group. The function returns

(a) the quotient group Q, and

(b) the natural homomorphism $f : G \to Q$.

Arbitrary quotients may be readily constructed in the case of the categories `GrpFP`, `GrpGPC`, `GrpPC` and `GrpAb`. However, in the case of permutation and matrix groups, currently the quotient group is constructed via its regular representation, so that the application of this operator is restricted to the case where the index of N in G is less than 2^{30}.

The second return value is the epimorphism from G to the resulting quotient group.

`G / N`

Given a (normal) subgroup N of the group G, construct the quotient of G by N.

If G is in category `GrpFP`, N is not checked to be normal in G. In fact, the returned group is the quotient of G by the normal closure of N in G. For all other categories of groups, passing a subgroup which fails to be normal causes a runtime error.

If G is a permutation or matrix group, the quotient group is constructed via its regular representation, so that the application of this operator is restricted to the case where the index of N in G is at most a million. The result returned need not be regular, as an attempt is made to reduce the degree of the result.

Example H57E7

Construction of the quotient of an abelian group, with a demonstration of the use of the natural homomorphism:

```
> G<[x]>, f := AbelianGroup< h, i, j, k | 8*h, 4*i, 6*j, 2*k - h >;
> T, n := quo< G | x[1] + 2*x[2] + 24*x[3], 16*x[3] >;
> T;
Abelian Group isomorphic to Z/2 + Z/16
Defined on 2 generators
Relations:
    4*T.1 = 0
    16*T.2 = 0
> n(x);
[
    2*T.1,
    T.1 + 12*T.2,
    T.2
]
> n(sub< G | x[1] + x[2] + x[3] >);
Abelian Group isomorphic to Z/16
Defined on 1 generator in supergroup T:
    $.1 = 3*T.1 + T.2
Relations:
    16*$.1 = 0
```

57.4 Standard Groups and Extensions

57.4.1 Construction of a Standard Group

A number of functions are provided which construct various standard groups. The effect of these functions is to construct the group on some standard set of generators. The group category of the result may be specified as an argument to the function.

`AbelianGroup(C, Q)`

`AbelianGroup(Q)`

Construct the abelian group defined by the sequence $Q = [n_1, \ldots, n_r]$ of positive integers. The function constructs the direct product of cyclic groups $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_r}$. In some categories, n_i may also be 0, denoting the infinite cyclic group $\mathbf{Z}$. If the single-argument version of the function is used, the group will be constructed in the category `GrpAb`; otherwise, its category will be C, where C may be `GrpAb`, `GrpFP`, `GrpGPC`, `GrpPC` or `GrpPerm`.

`AlternatingGroup(C, n)`

`AlternatingGroup(n)`

`Alt(C, n)`

`Alt(n)`

Construct the alternating group on n letters. If the single-argument version of the function is used, the group will be constructed in the category `GrpPerm`; otherwise, its category will be C, where C may be `GrpFP` or `GrpPerm`.

`CyclicGroup(C, n)`

`CyclicGroup(n)`

Construct the cyclic group of order n. If the single-argument version of the function is used, the group will be constructed in the category `GrpPerm`; otherwise, its category will be C, where C may be `GrpAb`, `GrpFP`, `GrpGPC`, `GrpPC` or `GrpPerm`.

`DihedralGroup(C, n)`

`DihedralGroup(n)`

Construct the dihedral group of order $2 * n$. If the single-argument version of the function is used, the group will be constructed in the category `GrpPerm`; otherwise, its category will be C, where C may be `GrpFP`, `GrpGPC`, `GrpPC` or `GrpPerm`.

`DicyclicGroup(n)`

`DicyclicGroup(A, a)`

The first intrinsic constructs the dicyclic group of order $4n$. The second, when given an abelian group A and an element a of order 2, constructs the associated dicyclic group generated by A and an x with $x^2 = a$ and $a^x = a^{-1}$ for all $x \in A$.

`SymmetricGroup(C, n)`

`SymmetricGroup(n)`

`Sym(GrpFin, n)`

`Sym(n)`

Construct the symmetric group on n letters. If the single-argument version of the function is used, the group will be constructed in the category `GrpPerm`; otherwise, its category will be C, where C may be `GrpFP` or `GrpPerm`.

`ExtraSpecialGroup(C, p, n : parameters)`

`ExtraSpecialGroup(p, n : parameters)`

Given a prime p and a small positive integer n, construct an extra-special group G of order p^{2n+1}. The isomorphism type of G can be selected using the parameter `Type` described below.

If the two-argument version of the function is used, the group will be constructed in the category `GrpPerm`; otherwise, its category will be C, where C may be `GrpFP`, `GrpGPC`, `GrpPC` or `GrpPerm`. If C is `GrpFP`, `GrpPC` or `GrpPerm`, the prime p must be small.

`Type` MonStgElt *Default :* "+"

Possible values for this parameter are "+" (default) and "−".

If `Type` is set to "+", the function returns for $p = 2$ the central product of n copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p.

If `Type` is set to "−", the function returns for $p = 2$ the central product of a quaternion group of order 8 and $n - 1$ copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p^2.

Example H57E8

(1) The abelian group $\mathbf{Z}_6 \times \mathbf{Z}_2 \times \mathbf{Z}_7$ in the category `GrpAb`:

```
> A := AbelianGroup([6, 2, 7]);
> A;
Abelian Group isomorphic to Z/2 + Z/42
Defined on 3 generators
Relations:
    6*A.1 = 0
    2*A.2 = 0
    7*A.3 = 0
```

(2) The alternating group on 6 letters as a permutation group:

```
> A6 := Alt(6);
> A6;
Permutation group A6 acting on a set of cardinality 6
Order = 360 = 2^3 * 3^2 * 5
```

```
    (1, 2)(3, 4, 5, 6)
    (1, 2, 3)
```

(3) The dihedral group of order 8 as a `GrpPC`:

```
> D8 := DihedralGroup(GrpPC, 4);
> D8;
GrpPC : D8 of order 8 = 2^3
PC-Relations:
  D8.2^2 = D8.3,
  D8.2^D8.1 = D8.2 * D8.3
```

(4) The symmetric group on 7 letters as a finitely presented group on generators a and b:

```
> S7<a, b> := SymmetricGroup(GrpFP, 7);
> S7;
Finitely presented group S7 on 2 generators
Relations
    a^7 = Id(S7)
    b^2 = Id(S7)
    (a * b)^6 = Id(S7)
    (a^-1 * b * a * b)^3 = Id(S7)
    (b * a^-2 * b * a^2)^2 = Id(S7)
    (b * a^-3 * b * a^3)^2 = Id(S7)
```

57.4.2 Construction of Extensions

`DirectProduct(G, H)`

Given two groups G and H belonging to the category C, construct the direct product of G and H as a group in C.

`DirectProduct(Q)`

Given a sequence Q of n groups belonging to the category C, construct the direct product $Q[1] \times Q[2] \times \ldots \times Q[n]$ as a group in the category C.

`SemidirectProduct(K, H, f:` *parameters*`)`

Given two groups K and H and a homomorphism $f : H \to \mathrm{Aut}(K)$, construct the semidirect product of K and H where the elements of H act on K via the map f. Return the semidirect product, and maps embedding H and K into the semidirect product.

`MaxDeg` RngIntElt *Default :* 1000000

The maximum degree permutation representation the algorithm will attempt.

`UseRegular` BoolElt *Default :* `false`

Setting `UseRegular` to true forces the algorithm to go via the regular representations of K and H.

Example H57E9

We define G to be the symmetric group of degree 4 and H to be the dihedral group of order 8. We then form the direct product of G and H.

```
> G := SymmetricGroup(4);
> H := DihedralGroup(3);
> D := DirectProduct(G, H);
> D;
Permutation group D acting on a set of cardinality 7
    (1, 2, 3, 4)
    (1, 2)
    (5, 6, 7)
    (5, 6)
> Order(D);
144
```

57.5 Transfer Functions Between Group Categories

Since certain group computations are possible or feasible only for particular group representations, it is often useful to transfer a group from one category to another. The functions in this section take a group and return a group isomorphic to it (or isomorphic to some related group) in another category.

`pQuotient(F, p, c: parameters)`

Given a group F in category `GrpFP`, a prime p and a positive integer c, construct the largest p-quotient G of F having lower exponent-p class at most c (or 127, if c is given as 0) as group in the category `GrpPC`. The function also returns the homomorphism from F to G.

The parameters are:

`Exponent` RngIntElt *Default :* 0

If `Exponent := m`, enforce the exponent law, $x^m = 1$, on the group.

`Metabelian` BoolElt *Default :* `false`

If `Metabelian := true`, then a consistent pcp is constructed for the largest metabelian p-quotient of F having lower exponent-p class at most c.

`Print` RngIntElt *Default :* 0

This parameter controls the volume of printing. By default its value is that returned by `GetVerbose("pQuotient")`, which is 0 unless it has been changed through use of `SetVerbose`. The effect is the following:

`Print := 0` : No output.

`Print := 1` : Report order of p-quotient at each class.

`Print := 2` : Report statistics and redundancy information about tails, consistency, collection of relations and exponent enforcement components of calculation.

`Print := 3` : Report in detail on the construction of each class.
Note that the presentation displayed is a *power-commutator* presentation (since this is the version stored by the p-quotient).

`Workspace` RngIntElt *Default :* 5000000

The amount of space requested for the p-quotient computation.

`CosetAction(G, H)`

Given a subgroup H of the group G, construct the permutation representation of G given by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The natural homomorphism $f : G \to L$;

(b) The induced permutation group L (the image of f);

(c) (if possible) The kernel K of the action (a subgroup of G).

If G is a finitely presented group, then K may be returned undefined.

The permutation representation is obtained by using the Todd-Coxeter procedure to construct the coset table for H in G. Note that G may be an infinite group: it is only necessary that the index of H in G be finite.

`CosetImage(G, H)`

Given a subgroup H of the group G, construct the image of G given by its action on the (right) coset space of H in G, returning it as a permutation group. (This is also the second return value of `CosetAction(G, H)`.)

`CosetKernel(G, H)`

Given a subgroup H of the group G, construct the kernel of G in its action on the (right) coset space of H in G. (This is also the third return value of `CosetAction(G, H)`.) This function may fail if G is a finitely presented group; it is only available when the index of H in G is very small.

`GPCGroup(G)`

Given a soluble group G, in the category `GrpPerm`, `GrpMat`, `GrpAb` or `GrpPC`, construct a polycyclic group P isomorphic to G. Currently G must be finite, if it is in the category `GrpMat`. In addition to returning P, the function returns an isomorphism $\phi : G \to P$.

`PCGroup(G)`

Given a finite soluble group G, in the category `GrpPerm`, `GrpMat`, `GrpAb` or `GrpGPC`, construct a group S given by a power-conjugate presentation, which is isomorphic to G. In addition to returning S, the function returns an isomorphism $\phi : G \to S$.

FPGroup(G: *parameters*)

StrongGenerators	BoolElt	*Default :* false
Random	BoolElt	*Default :* true
Max	RngIntElt	*Default :* 100
Run	RngIntElt	*Default :* 20

Given a group G, in the category GrpPerm, GrpMat, GrpGPC or GrpPC, construct a finitely presented group F isomorphic to G, by presenting the group on its given generators. For groups in the category GrpPerm and GrpMat, the Todd-Coxeter Schreier algorithm is used to construct the presentation and a choice of a presentation on the given generators or on the strong generators is available. In addition to returning F, the function returns an isomorphism $\phi : F \to G$, such that $\phi(F.i) = G.i$ for all i.

If the parameter StrongGenerators is set to true (GrpPerm and GrpMat only), the presentation will be constructed on the strong generators of G instead of the given generators. If strong generators are not already known for G, they will be constructed; in this case, the other parameters are also meaningful. The parameter Random with its associated parameters Max and Run may be used to apply the Random Schreier algorithm to construct a probable BSGS before commencing the construction of the presentation.

Example H57E10

We construct a finitely presented group G and a subgroup H, then find the permutation representation of G given by its action on the cosets of H. Since the induced permutation group L has the same order as G, the representation is faithful, and the homomorphism $f : G \to L$ is an isomorphism.

```
> G<a, b> := Group< a, b | a^3, b^3, (b * a)^4,
>     ((b^-1)^a * b^-1)^2 * b^a * b >;
> Order(G);
168
> H := sub< G | a^2 * b^2, (a * b)^2 >;
> Index(G, H);
7
> f, L := CosetAction(G, H);
> f;
Mapping from: GrpFP: G to GrpPerm: L
> L;
Permutation group L acting on a set of cardinality 7
    (1, 2, 3)(4, 7, 5)
    (1, 3, 4)(2, 5, 6)
> Order(L);
168
```

Example H57E11

A permutation representation of $\mathrm{Sp}(2,4)$.

```
> M := SymplecticGroup(2, 4);
> #M;
60
> Ms := sub< M | M.1 * M.2 >;
> Index(M, Ms);
12
> PG := CosetImage(M, Ms);
> PG;
Permutation group PG acting on a set of cardinality 12
    (1, 2, 4)(3, 5, 7)(6, 8, 10)(9, 11, 12)
    (1, 3, 2)(4, 6, 8)(5, 7, 9)(10, 12, 11)
> #PG;
60
```

Example H57E12

A finitely presented group isomorphic to $\mathrm{PSU}(3,3)$:

```
> G := PSU(3, 3);
> F<a, b>, phi := FPGroup(G);
> F;
Finitely presented group F on 2 generators
Relations
    a^8 = Id(F)
    b^8 = Id(F)
    (b * a^-1 * b)^3 = Id(F)
    b * a^-1 * b^-1 * a^-1 * b^-1 * a^-1 * b * a^-1 * b * a^-1 = Id(F)
    b^-1 * a^-2 * b^-1 * a^-2 * b^-1 * a^-1 * b^-2 * a^-1 = Id(F)
> phi(a) eq G.1 and phi(b) eq G.2;
true
```

57.6 Basic Operations

57.6.1 Accessing Group Information

The functions in this group provide access to basic information stored for a group G.

`G . i`

The i-th defining generator for G, if $i > 0$. If $i < 0$, then the inverse of the $-i$-th defining generator is returned. `G.0` is equivalent to `Identity(G)`.

`Generators(G)`

A set containing the defining generators for G.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for G.

`Generic(G)`

Given a group G in the category `GrpPerm` or `GrpMat`, return the generic group containing G, i.e., the largest group in which G is naturally embedded. The precise definition of generic group depends upon the category to which G belongs.

`Parent(g)`

The parent group G for the group element g.

Example H57E13

The Suzuki simple group $G = \mathrm{Sz}(8)$ is constructed. Its generic group is $\mathrm{GL}(4, K)$, where K is the finite field with 8 elements. The field K is constructed first, so that its generator may be given the printname z. Then the three generators of G are printed, in the standard order of indexing.

```
> K<z> := GF(2, 3);
> G := SuzukiGroup(8);
> Generic(G);
GL(4, GF(2, 3))
> Ngens(G);
3
> for i in [1..3] do
>    print "generator", i, G.i;
>    print "order", Order(G.i), "\r";
> end for;
generator 1
[  0   0   0   1]
[  0   0   1   0]
[  0   1   0   0]
[  1   0   0   0]
order 2

generator 2
```

```
[z^2   0   0   0]
[  0 z^6   0   0]
[  0   0   z   0]
[  0   0   0 z^5]
order 7

generator 3
[  1   0   0   0]
[z^2   1   0   0]
[  0   z   1   0]
[z^5 z^3 z^2   1]
order 4
```

`Orbit(G, M, x)`

Given a finitely generated group G that acts on the parent structure of x through the map (or user defined function) M, compute the orbit of x under G. Thus, for every generator g of G, $M(g)$ must return a function that can be applied to x or any other element in the parent of x.

If the orbit is infinite, this process will eventually run out of memory.

`OrbitClosure(G, M, S)`

Given a finitely generated group G acting on the universe of S through the map or user defined function M, compute the smallest subset T containing S that is G-invariant. Thus, for every generator g of G, $M(g)$ must return a function that can be applied to an arbitrary element in the universe of S.

If the orbit closure is infinite, this process will eventually run out of memory.

57.7 Operations on the Set of Elements

57.7.1 Order and Index Functions

`Order(G)`

`#G`

The order of the group G as an integer. If the order is not currently known, it will be computed. Computing the order may fail for groups in the category `GrpFP`; cf. Chapter 70.

`FactoredOrder(G)`

The order of the finite group G returned as a factored integer. The factorization is returned in the form of a sequence Q which is defined as follows: If $\#G = p_1^{e_1} \dots p_n^{e_n}, \quad e_i > 0$, then Q will be the integer sequence $[\langle p_1, e_1\rangle, \dots, \langle p_n, e_n\rangle]$. If the orders of G is not known, it will be computed. Computing the order may fail for groups in the category `GrpFP`; cf. Chapter 70.

`Index(G, H)`

The index of the subgroup H in the group G. The index is returned as an integer. Computing the index may fail for groups in the category `GrpFP`; cf. Chapter 70.

`FactoredIndex(G, H)`

The index of the subgroup H in the group G. H must have finite index in G. The index is returned as a factored integer. The format is the same as for `FactoredOrder`. Computing the index may fail for groups in the category `GrpFP`; cf. Chapter 70.

Example H57E14

Exploration of the order and index functions for a finitely presented group and its subgroup:

```
> Q<s,t,u>, h := Group< s, t, u |
>     t^2, u^17, s^2 = t^s = t, u^s = u^16, u^t = u >;
> Order(Q);
68
> FactoredOrder(Q);
[ <2, 2>, <17, 1> ]
> S := sub< Q | t*s^2, u^4 >;
> Index(Q, S);
4
> #S;
17
```

57.7.2 Membership and Equality

`g in G`

Given a group element g and a group G, return `true` if g is an element of G, `false` otherwise.

`g notin G`

Given a group element g and a group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G`

Given a group G and a set S of group elements belonging to a group H, where G and H belong the same generic group, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G`

Given a group G and a set S of group elements belonging to a group H, where G and H belong the same generic group, return `true` if S is not a subset of G, `false` otherwise.

`H subset G`

Given groups G and H belonging to the same generic group, return `true` if H is a subgroup of G, `false` otherwise.

`H notsubset G`

Given groups G and H belonging to the same generic group, return `true` if H is not a subgroup of G, `false` otherwise.

`H eq G`

Given groups G and H belonging to the same generic group, return `true` if G and H are the same group, `false` otherwise.

`H ne G`

Given groups G and H belonging to the same generic group, return `true` if G and H are distinct groups, `false` otherwise.

57.7.3 Set Operations

`NumberingMap(G)`

Given a finite group G in the category `GrpPerm`, `GrpMat`, `GrpPC` or `GrpAb`, return a bijective mapping from the group G onto the set of integers $\{1 \ldots |G|\}$. The actual mapping depends upon the particular representation of G.

`Representative(G)`

`Rep(G)`

An element chosen from the group G.

Example H57E15

We use the function `NumberingMap` to construct the multiplication table for the dihedral group of order 12.

```
> G := DihedralGroup(6);
> f := NumberingMap(G);
> [ [ f(x*y) : y in G ] : x in G ];
[
    [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ],
    [ 2, 3, 4, 5, 6, 1, 12, 7, 8, 9, 10, 11 ],
    [ 3, 4, 5, 6, 1, 2, 11, 12, 7, 8, 9, 10 ],
    [ 4, 5, 6, 1, 2, 3, 10, 11, 12, 7, 8, 9 ],
    [ 5, 6, 1, 2, 3, 4, 9, 10, 11, 12, 7, 8 ],
    [ 6, 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 7 ],
    [ 7, 8, 9, 10, 11, 12, 1, 2, 3, 4, 5, 6 ],
    [ 8, 9, 10, 11, 12, 7, 6, 1, 2, 3, 4, 5 ],
    [ 9, 10, 11, 12, 7, 8, 5, 6, 1, 2, 3, 4 ],
    [ 10, 11, 12, 7, 8, 9, 4, 5, 6, 1, 2, 3 ],
```

```
    [ 11, 12, 7, 8, 9, 10, 3, 4, 5, 6, 1, 2 ],
    [ 12, 7, 8, 9, 10, 11, 2, 3, 4, 5, 6, 1 ]
]
```

57.7.4 Random Elements

`Random(G: parameters)`

`Short` BoolElt *Default :* `false`

A randomly chosen element for the group G. If a representation of the carrier set of G has already been created, then the element chosen will be genuinely random. If such a representation has not yet been created, then the *random* element is chosen by multiplying out a *random* word in the generators. Since it is not usually practical to choose words long enough to properly sample the elements of G, the element returned will usually be biased. The boolean-valued parameter `Short` is used in this situation to indicate that a short word will suffice. Thus, if `Random` is invoked with `Short` assigned the value `true` then the element is constructed using a short word.

Example H57E16

We illustrate the use of the function `Random` using the wreath product of the symmetric group of degree 4 and the cyclic group of order 6.

```
> G := WreathProduct(Sym(4), CyclicGroup(6));
> G;
Permutation group G acting on a set of cardinality 24
    (1, 5, 9, 13, 17, 21)(2, 6, 10, 14, 18, 22) (3, 7, 11, 15, 19, 23)
        (4, 8, 12, 16, 20, 24)
    (1, 2, 3, 4)
    (1, 2)
> Order(G);
1146617856
> Random(G);
(1, 17, 12, 4, 18, 10, 3, 20, 9, 2, 19, 11)(5, 22, 13, 6, 21, 15)
    (7, 24, 16)(8, 23, 14)
// We display the cycle structures of 10 random elements of G
> R := [ CycleStructure(Random(G)) : i in [1..10] ];
> R;
[
    [ <6, 1>, <3, 6> ],
    [ <9, 1>, <6, 2>, <3, 1> ],
    [ <9, 2>, <3, 2> ],
    [ <12, 1>, <9, 1>, <3, 1> ],
    [ <18, 1>, <6, 1> ],
    [ <18, 1>, <6, 1> ],
    [ <12, 1>, <6, 2> ],
```

```
    [ <6, 3>, <2, 3> ],
    [ <6, 1>, <4, 3>, <2, 3> ],
    [ <6, 3>, <3, 2> ]
]
```

`RandomProcess(G)`

`RandomProcessWithWords(G)`

`RandomProcessWithValues(G, Q)`

`RandomProcessWithWordsAndValues(G, Q)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 50
`WordGroup`	GrpSLP	*Default :*

Create a process to generate randomly chosen elements from the group G. The process uses a variant of the product-replacement method similar to the *Rattle* method of [LGM02]. The generating set stored in the process has N elements, where N is the maximum of the specified value for `Slots` and `Ngens`$(G)+1$. Initially they are the generators of G repeated as necessary and the accumulator is the identity. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call returns the current value of the accumulator, modifying the generating set as for product-replacement, and modifying the accumulator by multiplying by the new member of the generating set. Setting `Scramble := m` causes m such operations to be performed before the process is returned.

The functions with words and values create a process that returns extra group elements for each call. A process created with words returns, as second return value for each call to `Random(P)`, the `GrpSLPElt` describing the random element returned as a straight-line program in the group generators. The parameter `WordGroup` may be used to specify a particular group for the words to be elements of.

A process created with values takes as input a sequence of group elements Q giving the values assigned to each generator of G. The second value returned is the result of computing in parallel with these values as with the generators of G. In particular, if the elements of Q are homomorphic images of the generators of G, then the second return value from `Random(P)` will be the image of the first under this homomorphism.

A process created with words and values does all of the above, with the three return values of `Random(P)` being a random element of G, the straight-line program and the value.

The use of this function on a finitely-presented group G is not recommended. Since there is no reduction of words, the random elements generated may be extremely long.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the finite group G, construct a random element of G by forming a random product over the expanded generating set constructed when the process was created. For large permutation or matrix groups for which a BSGS is not known, this function should be used in preference to `Random(G)`.

If the process was created with words or values then there will be second and third return values as described under `RandomProcess` above.

`InitialiseProspector(G:`*parameters*`)`

`InitialiseProspector(G:`*parameters*`)`

Initialise a product-replacement prospector for the given group. This is an extension of the product-replacement algorithm that searches for an element $x \in G$ such that some predicate is true for this element. The prospector aims to find elements x so that the corresponding straight-line program for x is short. Statistical tests and various heuristics are used to achieve this.

Generally, output from product-replacement with short straight-line programs is not very random. Prospector aims to run product-replacement until the output looks random, then start a search for the element wanted. At all times, if the output starts to look non-random, or word lengths grow too far without finding an element, the prospector may return to a previous state of product replacement and try again, searching in a different direction. The statistical tests are used to make concrete the notion of "looks random". For permutation groups the test used is based on number of cycles of the element. For matrix groups the test statistic is the number of factors of the characteristic polynomial.

`Prospector(G, f:`*parameters*`)`

Run an initialised prospector for group G to find $x \in G$ such that $f(x)$ is true. The first return value gives the success or failure of the search. If this value is true, then the second and third return values are x and a straight-line program giving x in terms of the group generators. The parameter `MaxTries` may be set to limit the number of random selections made by the prospector when attempting to find x.

Example H57E17

We find a random pair of generators for the symmetric group of degree 300 and use a random process to find an element which is a 300-cycle as a straight-line program in the generators. The proportion of such elements is 1 in 300, so we expect the program to have length 600.

```
> SetSeed(1);
> S := Sym(300);
> repeat G := sub<S|Random(S),Random(S)>;
> until IsSymmetric(G);
> P := RandomProcessWithWords(G);
> repeat x,w := Random(P);
```

```
> until CycleStructure(x) eq [<300,1>];
> #w;
936
```

Note that the group S, known to be a symmetric group, has an efficient uniform random element generator available as above. The word length was somewhat longer than the expected value. Now we set up a prospector and use it to search for an element of the same cycle structure. The defining word for the new element should be shorter than the expected 600.

```
> InitialiseProspector(G);
true
> test := func<x|CycleStructure(x) eq [<300,1>]>;
> a,x,w := Prospector(G, test);
> a;
true
> #w;
206
> Evaluate(w, [G.1,G.2]) eq x;
true
```

57.7.5 Action on a Coset Space

`CosetTable(G, H)`

The (right) coset table for G over subgroup H relative to its defining generators.

`#CosetTable(G, f)`

The coset table for G corresponding to the permutation representation f of G, where f is a homomorphism of G onto a transitive permutation group.

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a group G and a subgroup H of G, this function returns

(a) An indexed set of elements T of G forming a right transversal for G over H;

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and $g \in G$, ϕ is defined by $\phi(g) = t_i$, where $g \in Ht_i$.

`CosetAction(G, H)`

Given a subgroup H of the group G, construct the permutation representation of G given by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The natural homomorphism $f : G \to L$;

(b) The induced group L;

(c) The kernel K of the action (a subgroup of G).

Note that G may be any type of group. If G is a finitely presented group, then K may be returned undefined.

`CosetImage(G, H)`

Given a subgroup H of the group G, construct the image L of G given by the action of G on the set of (right) cosets of H in G.

`CosetKernel(G, H)`

Given a subgroup H of the group G, construct the kernel of the action of G on the set of (right) cosets of H in G.

57.8 Standard Subgroup Constructions

Some functions described in this section may not exist or may have restrictions for some categories of groups. Details can be found in the chapters on the individual categories.

`H ^ g`

`Conjugate(H, g)`

Construct the conjugate $g^{-1}Hg$ of the group H by the element g. The group H and the element g must belong to the same generic group.

`H meet K`

Given groups H and K which belong to the same symmetric group, construct the intersection of H and K.

`CommutatorSubgroup(G, H, K)`

`CommutatorSubgroup(H, K)`

Given groups H and K, both subgroups of the group G, construct the commutator subgroup of H and K in the group G. If K is a subgroup of H, then the group G may be omitted.

`Centralizer(G, g)`

`Centraliser(G, g)`

Construct the centralizer of the element g in the group G.

`Centralizer(G, H)`
`Centraliser(G, H)`

Construct the centralizer of the group H in the group G.

`Core(G, H)`

Given a subgroup H of the group G, construct the maximal normal subgroup of G that is contained in the subgroup H.

`H ^ G`
`NormalClosure(G, H)`

Given a subgroup H of the group G, construct the normal closure of H in G.

`Normalizer(G, H)`
`Normaliser(G, H)`

Given a subgroup H of the group G, construct the normalizer of H in G.

`pCore(G, p)`

Given a group G and a prime p dividing the order of G, construct the maximal normal p-subgroup of G.

`SylowSubgroup(G, p)`
`Sylow(G, p)`

Given a group G and a prime p, construct a Sylow p-subgroup of G.

57.8.1 Abstract Group Predicates

Some functions described in this section may not exist or may have restrictions for some categories of groups. Details can be found in the chapters on the individual categories.

`IsAbelian(G)`

Returns `true` if the group G is abelian, `false` otherwise.

`IsCyclic(G)`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group G is elementary abelian, `false` otherwise.

`IsCentral(G, H)`

Return `true` if the subgroup H of the group G lies in the centre of G, `false` otherwise.

`IsConjugate(G, g, h)`

Given a group G and elements g and h belonging to G, return the value `true` if g and h are conjugate in G. The function returns a second value if the elements are conjugate: an element k which conjugates g into h.

`IsConjugate(G, H, K)`

Given a group G and subgroups H and K belonging to G, return the value `true` if H and K are conjugate in G. The function returns a second value if the subgroups are conjugate: an element z which conjugates H into K.

`IsExtraSpecial(G)`

Given a group G is a p-group G, return `true` if G is extra-special, `false` otherwise.

`IsMaximal(G, H)`

Returns `true` if the subgroup H of the group G is a maximal subgroup of G. This function is evaluated by constructing the permutation representation of G on the cosets of H and testing this representation for primitivity. For this reason, the use of `IsMaximal` should be avoided if the index of H in G exceeds a one hundred thousand.

`IsNilpotent(G)`

Return `true` if the group G is nilpotent, `false` otherwise.

`IsNormal(G, H)`

Return `true` if the subgroup H of the group G is a normal subgroup of G, `false` otherwise.

`IsPerfect(G)`

Return `true` if the group G is perfect, `false` otherwise.

`IsSelfNormalizing(G, H)`
`IsSelfNormalising(G, H)`

Return `true` if the subgroup H of the group G is self-normalizing in G, `false` otherwise.

`IsSimple(G)`

Return `true` if the group G is simple, `false` otherwise.

`IsSoluble(G)`
`IsSolvable(G)`

Return `true` if the group G is soluble, `false` otherwise.

`IsSpecial(G)`

Given a p-group G, return `true` if G is special, `false` otherwise.

`IsSubnormal(G, H)`

Return `true` if the subgroup H of the group G is subnormal in G, `false` otherwise.

`IsTrivial(G)`

Return `true` if G is trivial, `false` otherwise.

57.9 Characteristic Subgroups and Normal Structure

57.9.1 Characteristic Subgroups and Subgroup Series

Some functions described in this section may not exist or may have restrictions for some categories of groups. Details can be found in the chapters on the individual categories.

`Centre(G)`
`Center(G)`

Construct the centre of the group G.

`Hypercentre(G)`
`Hypercenter(G)`

Construct the hypercentre of the group G (the stationary term of the upper central series).

`DerivedLength(G)`

The derived length of G. If G is non-soluble, the function returns the number of terms in the series terminating with the soluble residual.

`DerivedSeries(G)`

The derived series of the group G. The series is returned as a sequence of subgroups.

`DerivedSubgroup(G)`
`DerivedGroup(G)`

The derived subgroup of the group G.

`FittingSubgroup(G)`

The Fitting subgroup of the group G.

`FrattiniSubgroup(G)`

Given a group G that is a p-group, return the Frattini subgroup.

`JenningsSeries(G)`

Given a p-group G, return the Jennings series for G. The series is returned as a sequence of subgroups.

`LowerCentralSeries(G)`

The lower central series of G. The series is returned as a sequence of subgroups.

`NilpotencyClass(G)`

The nilpotency class of the group G. If the group is not nilpotent, the value -1 is returned.

`H ^ G`

`NormalClosure(G, H)`

The normal closure of the subgroup H of group G.

`NormalLattice(G)`

The normal subgroups of G arranged as a lattice.

`NormalSubgroups(G)`

The normal subgroups of G.

`pCentralSeries(G, p)`

Given a soluble group G, and a prime p dividing $|G|$, return the lower p-central series for G. The series is returned as a sequence of subgroups.

`Radical(G)`

The maximal normal solvable subgroup of the group G.

`SolubleResidual(G)`

`SolvableResidual(G)`

The solvable residual of the group G.

`SubnormalSeries(G, H)`

Given a group G and a subnormal subgroup H of G, return a sequence of subgroups commencing with G and terminating with H, such that each subgroup is normal in the previous one. If H is not subnormal in G, the empty sequence is returned.

`UpperCentralSeries(G)`

The upper central series of G. The series is returned as a sequence of subgroups commencing with the trivial subgroup. Since the algorithm used requires the conjugacy classes of G, this function is much more restricted in its range of application than `DerivedSeries` and `LowerCentralSeries`.

f	Family name
1	A(d, q)
2	B(d, q)
3	C(d, q)
4	D(d, q)
5	G$(2, q)$
6	F$(4, q)$
7	E$(6, q)$
8	E$(7, q)$
9	E$(8, q)$
10	2A(d, q)
11	2B$(2, q)$
12	2D(d, q)
13	3D$(4, q)$
14	2G$(2, q)$
15	2F$(4, q)$
16	2E$(6, q)$
17	Alternating(d)
18	Sporadic group — see Table 2.
19	Cyclic(q)

Table 1: Family numbers and names

d	Group name
1	M_{11}
2	M_{12}
3	M_{22}
4	M_{23}
5	M_{24}
6	J_1
7	HS
8	J_2
9	MCL
10	SUZ
11	J_3
12	CO_1
13	CO_2
14	CO_3
15	HE
16	$M(22)$
17	$M(23)$
18	$M(24)$
19	LY
20	RU
21	ON
22	TH
23	HA
24	BM
25	M
26	J_4

Table 2: Sporadic groups

57.9.2 The Abstract Structure of a Group

`CompositionFactors(G)`

Given a finite group G in the category `GrpPerm`, `GrpMat`, `GrpPC` of `GrpAb`, return a sequence S of tuples that represent the composition factors of G, ordered according to some composition series of G. Each tuple is a triple of integers f, d, q that defines the isomorphism type of the corresponding composition factor. A triple $< f, d, q >$ describes a simple group as follows. The integer f defines the family to which the group belongs, and d and q are the parameters of the family. The length of the sequence S is the number of composition factors of G. The families are listed in Tables 1 and 2 on page 1495.

`AbelianInvariants(G)`

`Invariants(G)`

Given an abelian group G in the category `GrpPerm`, `GrpMat`, `GrpPC` of `GrpAb`, return a sequence Q containing the types of each p-primary component of G.

`AbelianBasis(G)`

Given an abelian group G in the category `GrpPerm`, `GrpPC` of `GrpAb`, return sequences B and I where I contains the types of each p-primary component of G and B contains corresponding elements of G which have the order given and generate G.

57.10 Conjugacy Classes of Elements

There are three aspects of the conjugacy problem for elements: determining whether two elements are conjugate in a group G, determining a set of representatives for the conjugacy classes of elements of G, and listing all the elements in a particular class of G. The algorithms used depend on the category of G. If G is in category `GrpPerm` or `GrpMat`, conjugacy is determined by means of a backtrack search over base-images. If G is in category `GrpPC`, testing conjugacy is performed by transforming each element into a standard representative of its conjugacy class by an orbit-stabilizer process that works down a sequence of increasing quotients of G. Conjugacy testing for a group G in category `GrpGPC` is only possible if G is nilpotent. In this case, an algorithm by E. Lo [Lo98] is used.

Some functions described in this section may not exist or may have restrictions for some categories of groups. Details can be found in the chapters on the individual categories.

`Class(H, x)`

`Conjugates(H, x)`

Given a group H and an element x belonging to a group K such that H and K are subgroups of the same symmetric group, this function returns the set of conjugates of x under the action of H. If $H = K$, the function returns the conjugacy class of x in H.

`ClassMap(G:` *parameters*`)`

Given a group G, construct the conjugacy classes and the class map f for G. For any element x of G, $f(x)$ will be the index of the conjugacy class of x in the sequence returned by the `Classes` function.

If G is a permutation group, the construction may be controlled using the parameters `Orbits`, `WeakLimit` and `StrongLimit`. If the parameter `Orbits` is set `true`, the classes are computed as orbits of elements under conjugation and the class map is stored as a list of images of the elements of G (a list of length $|G|$). This option gives fast evaluation of the class map but is practical only in the case of very small groups. With `Orbits := false`, `WeakLimit` and `StrongLimit` are used to control the random classes algorithm (see function `Classes`).

`ConjugacyClasses(G: parameters)`

`Classes(G: parameters)`

`WeakLimit`	RngIntElt	*Default :* 200
`StrongLimit`	RngIntElt	*Default :* 500
`Reps`	[GrpFinElt]	*Default :*
`Al`	MonStg	*Default :*

Construct a set of representatives for the conjugacy classes of G. The classes are returned as a sequence of triples containing the element order, the class length and a representative element for the class. The parameters `Reps` and `Al` enable the user to select the algorithm that is to be used when G is a permutation or matrix group.
`Reps := Q`: Create the classes of G by assuming that Q is a sequence of class representatives for G. The orders and lengths of the classes will be computed and checked for consistency.
`Al := "Action"`: Create the classes of G by computing the orbits of the set of elements of G under the action of conjugation. This option is only feasible for small groups.
`Al := "Random"`: Construct the conjugacy classes of elements for a permutation or matrix group G using an algorithm that searches for representatives of all conjugacy classes of G by examining a random selection of group elements and their powers. The behaviour of this algorithm is controlled by two associated optional parameters `WeakLimit` and `StrongLimit`, whose values are positive integers n_1 and n_2, say. Before describing the effect of these parameters, some definitions are needed: A mapping $f : G \to I$ is called a *class invariant* if $f(g) = f(g^h)$ for all $g, h \in G$. For permutation groups, the cycle structure of g is a readily computable class invariant. In matrix groups, the primary invariant factors are used where possible, or the characteristic or minimal polynomials otherwise. Two elements g and h are said to be *weakly conjugate* with respect to the class invariant f if $f(g) = f(h)$. By definition, conjugacy implies weak conjugacy, but the converse is false. The random algorithm first examines n_1 random elements and their powers, using a test for weak conjugacy. It then proceeds to examine a further n_2 random elements and their powers, using a test for ordinary conjugacy. The idea behind this strategy is that the algorithm should attempt to find as many classes as possible using the very cheap test for weak conjugacy, before employing the more expensive ordinary conjugacy test to recognize the remaining classes.
`Al := "Extend"`: Construct the conjugacy classes of G by first computing classes in a quotient G/N and then extending these classes to successively larger quotients G/H until the classes for $G/1$ are known. More precisely, a maximal series of subgroups $1 = G_0 < G_1 < \cdots < G_r = R < G$ is computed such that R is the (solvable) radical of G and G_{i+1}/G_i is elementary abelian. A representation of G/R is computed using an algorithm of Derek Holt and its classes computed and represented as elements of G. To extend to the next larger quotient, a group is computed from each class which acts on the transversal. Each distinct orbit in

that action gives rise to a new class. To compute the classes of G/R, the default algorithm (excluding the extension method) is used. The same set of parameters is passed on, so you can control limits in the random classes method if it is chosen. The limitations of the algorithm are that R may be trivial, in which case nothing is done except to call a different algorithm, or one or more of the sections may be so large as to prohibit computing the action on the transversal. This algorithm is currently only available for permutation groups.

`ClassRepresentative(G, x)`

Given a group G for which the conjugacy classes are known and an element x of G, return the designated representative for the conjugacy class of G containing x.

`IsConjugate(G, g, h)`

Given a group G and elements g and h belonging to G, return the value `true` if g and h are conjugate in G. The function returns a second value if the elements are conjugate: an element k which conjugates g into h.

`IsConjugate(G, H, K)`

Given a group G and subgroups H and K belonging to G, return the value `true` if G and H are conjugate in G. The function returns a second value if the subgroups are conjugate: an element z which conjugates H into K.

`Exponent(G)`

The exponent of the group G.

`NumberOfClasses(G)`
`Nclasses(G)`

The number of conjugacy classes of elements for the group G.

`PowerMap(G)`

Given a group G, construct the power map for G. Suppose that the order of G is m and that G has r conjugacy classes. When the classes are determined by MAGMA, they are numbered from 1 to r. Let C be the set of class indices $\{1, \ldots, r\}$. The *power map* f for G is the mapping

$$f : C \times \mathbf{Z} \to C$$

where the value of $f(i, j)$ for $i \in C$ and $j \in \mathbf{Z}$ is the number of the class which contains x_i^j, where x_i is a representative of the i-th conjugacy class.

Example H57E18

The conjugacy classes of the Mathieu group M_{11} can be constructed as follows:

```
> M11 := sub<Sym(11) | (1,10)(2,8)(3,11)(5,7), (1,4,7,6)(2,11,10,9)>;
> Classes(M11);
Conjugacy Classes of group M11
------------------------------
[1]     Order 1       Length 1
        Rep Id(M11)

[2]     Order 2       Length 165
        Rep (3, 10)(4, 9)(5, 6)(8, 11)

[3]     Order 3       Length 440
        Rep (1, 2, 4)(3, 5, 10)(6, 8, 11)

[4]     Order 4       Length 990
        Rep (3, 6, 10, 5)(4, 8, 9, 11)

[5]     Order 5       Length 1584
        Rep (1, 3, 6, 2, 8)(4, 7, 10, 9, 11)

[6]     Order 6       Length 1320
        Rep (1, 11, 2, 6, 4, 8)(3, 10, 5)(7, 9)

[7]     Order 8       Length 990
        Rep (1, 4, 5, 6, 2, 7, 11, 10)(8, 9)

[8]     Order 8       Length 990
        Rep (1, 7, 5, 10, 2, 4, 11, 6)(8, 9)

[9]     Order 11      Length 720
        Rep (1, 11, 9, 10, 4, 3, 7, 2, 6, 5, 8)

[10]    Order 11      Length 720
        Rep (1, 9, 4, 7, 6, 8, 11, 10, 3, 2, 5)
```

57.11 Conjugacy Classes of Subgroups

MAGMA contains a new algorithm for computing representatives of the conjugacy classes of subgroups. Let R denote the maximal normal soluble subgroup of the finite group G. The algorithm first constructs representatives for the conjugacy classes of subgroups of $Q = G/R$, and then successively extends these to larger and larger quotients of G until G itself is reached. If G is soluble, then Q is trivial and so its subgroups are known. If G is non-soluble, we attempt to locate the quotient in a database of groups with trivial Fitting subgroup. This database contains all such groups of order up to 216 000, and all such which are perfect of order up to 1 000 000. If Q is found then either all its subgroups, or its maximal subgroups are read from the database. (In some cases only the maximal subgroups are stored.) If Q is not found then we attempt to find the maximal subgroups of Q using a method of Derek Holt. For this to succeed all simple factors of the socle of Q must be found in a second database which currently contains all simple groups of order less than 1.6×10^7, as well as M_{24}, HS, J_3, McL, $Sz(32)$ and $L_6(2)$. There are also special routines to handle numerous other groups. These include: A_n for $n \leq 999$, $L_2(q)$, $L_3(q)$, $L_4(q)$ and $L_5(q)$ for all q, $S_4(q)$, $U_3(q)$ and $U_4(q)$ for all q, $L_d(2)$ for $d \leq 14$, and the following groups: $L_6(3)$, $L_7(3)$, $U_6(2)$, $S_8(2)$, $S_{10}(2)$, $O_8^{\pm}(2)$, $O_{10}^{\pm}(2)$, $S_6(3)$, $O_7(3)$, $O_8^-(3)$, $G_2(4)$, $G_2(5)$, ${}^3D_4(2)$, ${}^2F_4(2)'$, Co_2, Co_3, He, Fi_{22}.

If we have only maximal subgroups of Q, and more are required, we apply the algorithm recursively to the maximal subgroups to determine all subgroups of Q. This may take some time.

57.11.1 Conjugacy Classes of Subgroups

In this section we describe the functions that allow a user to create representatives of the conjugacy classes of subgroups, possibly subject to conditions. The main function, `Subgroups`, finds representatives for conjugacy classes of subgroups subject to certain user-supplied conditions on the order. The alternative functions `ElementaryAbelianSubgroups` and `AbelianSubgroups`, `CyclicSubgroups`, `NilpotentSubgroups`, `SolubleSubgroups`, `PerfectSubgroups`, `NonsolvableSubgroups`, `SimpleSubgroups` and `RegularSubgroups` allow the user to construct particular classes of subgroups.

Most of the features described in this section are currently only available for groups in the category `GrpPerm`, `GrpMat` or `GrpPC`.

`SubgroupClasses(G: parameters)`

`Subgroups(G: parameters)`

Representatives for the conjugacy classes of subgroups for the group G. The subgroups are returned as a sequence of records where the i-th record contains:

(a) A representative subgroup H for the i-th conjugacy class (field name `subgroup`).

(b) The order of the subgroup (field name `order`).

(c) The number of subgroups in the class (field name `length`).

(d) [Optionally] A presentation for H (field name `presentation`).

`Presentation` BOOLELT *Default :* `false`

`Presentation := true`: In the case in which G is a permutation group, construct a presentation for each subgroup.

`OrderEqual` RngIntElt *Default :*

`OrderEqual := n`: Only construct subgroups having order equal to n.

`OrderDividing` RngIntElt *Default :*

`OrderDividing := n`: Only construct subgroups having order dividing n.

`IsNormal` BoolElt *Default :* `false`

`IsNormal := true`: Only construct normal subgroups.

`IsRegular` BoolElt *Default :* `false`

`IsRegular := true`: In the case in which G is a permutation group, only construct regular subgroups.

`LayerSizes` SeqEnum *Default :* see below

`LayerSizes := [ 2, 5, 3, 4, 7, 3, 11, 2, 17, 1]` is equivalent to the default. When constructing an Elementary Abelian series for the group, attempt to split 2-layers of size gt 2^5, 3-layers of size gt 3^4, etc. The implied exponent for 13 is 2 and for all primes greater than 17 the exponent is 1.

`ElementaryAbelianSubgroups(G: parameters)`

Representatives for the conjugacy classes of elementary abelian subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`AbelianSubgroups(G: parameters)`

Representatives for the conjugacy classes of abelian subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`CyclicSubgroups(G: parameters)`

Representatives for the conjugacy classes of cyclic subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`NilpotentSubgroups(G: parameters)`

Representatives for the conjugacy classes of nilpotent subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`SolubleSubgroups(G: parameters)`

`SolvableSubgroups(G: parameters)`

Representatives for the conjugacy classes of solvable subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`NonsolvableSubgroups(G: parameters)`

Representatives for the conjugacy classes of nonsolvable subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`PerfectSubgroups(G: parameters)`

Representatives for the conjugacy classes of perfect subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`SimpleSubgroups(G: parameters)`

Representatives for the conjugacy classes of non-abelian simple subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`RegularSubgroups(G: parameters)`

Representatives for the conjugacy classes of regular subgroups for the permutation group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`SetVerbose("SubgroupLattice", i)`

Turn on verbose printing for the subgroup algorithm. The level i can be 2 for maximal printing or 1 for moderate printing. The algorithm works down an elementary abelian series of the group and at each level, the possible extensions of each subgroup are listed.

`Class(G, H)`

`Conjugates(G, H)`

The G-conjugacy class of subgroups containing the group H.

Example H57E19

We construct the conjugacy classes of subgroups for the dihedral group of order 12.

```
> G := DihedralGroup(6);
> S := Subgroups(G);
> S;
Conjugacy classes of subgroups
------------------------------
[ 1]    Order 1            Length 1
        Permutation group acting on a set of cardinality 6
        Order = 1
          Id($)
[ 2]    Order 2            Length 3
        Permutation group acting on a set of cardinality 6
          (2, 6)(3, 5)
[ 3]    Order 2            Length 3
        Permutation group acting on a set of cardinality 6
          (1, 4)(2, 3)(5, 6)
[ 4]    Order 2            Length 1
        Permutation group acting on a set of cardinality 6
          (1, 4)(2, 5)(3, 6)
[ 5]    Order 3            Length 1
        Permutation group acting on a set of cardinality 6
          (1, 5, 3)(2, 6, 4)
[ 6]    Order 4            Length 3
        Permutation group acting on a set of cardinality 6
          (2, 6)(3, 5)
          (1, 4)(2, 5)(3, 6)
[ 7]    Order 6            Length 1
        Permutation group acting on a set of cardinality 6
          (1, 5, 3)(2, 6, 4)
          (1, 4)(2, 5)(3, 6)
[ 8]    Order 6            Length 1
        Permutation group acting on a set of cardinality 6
          (2, 6)(3, 5)
          (1, 5, 3)(2, 6, 4)
[ 9]    Order 6            Length 1
        Permutation group acting on a set of cardinality 6
          (1, 4)(2, 3)(5, 6)
          (1, 5, 3)(2, 6, 4)
[10]    Order 12           Length 1
        Permutation group acting on a set of cardinality 6
          (2, 6)(3, 5)
          (1, 5, 3)(2, 6, 4)
          (1, 4)(2, 5)(3, 6)
> // We extract the representative subgroup for class 7
> h := S[7]`subgroup;
> h;
```

```
Permutation group h acting on a set of cardinality 6
    (1, 3, 5)(2, 4, 6)
    (1, 4)(2, 5)(3, 6)
```

57.11.2 The Poset of Subgroup Classes

In addition to finding representatives for conjugacy classes of subgroups, MAGMA allows the user to create the poset L of subgroup classes. The elements of the poset correspond to the conjugacy classes of subgroups. Two lattice elements a and b are joined by an edge if either some subgroup of the conjugacy class a is a maximal subgroup of some subgroup of conjugacy class b or vice-versa. The elements of L are called *subgroup-poset elements* and are numbered from 1 to n, where n is the cardinality of L. Various functions allow the user to identify maximal subgroups, normalizers, centralizers and other relatives in the lattice. Given an element e of L, one can easily create the subgroup H of G corresponding to e and one can also create the element of L corresponding to a subgroup of G.

The features described in this section are currently only available for groups in the category `GrpPerm` or `GrpPC`.

57.11.2.1 Creating the Poset of Subgroup Classes

`SubgroupLattice(G)`

Create the poset L of subgroup classes of G.

`Properties` BOOLELT *Default :* `false`

`Properties := true`: As the subgroup classes are put into the poset, record their abstract type, i.e., elementary abelian, abelian, nilpotent, soluble, simple or perfect.

`Centralizers` BOOLELT *Default :* `false`

`Centralizers := true`: As each subgroup class e is put into the poset, record the class in which the centralizers of the subgroups of e lie.

`Normalizers` BOOLELT *Default :* `false`

`Normalizers := true`: As each subgroup class e is put into the poset, record the class in which the normalizers of the subgroups of e lie.

Example H57E20

We create the subgroup poset for the group ASL$(2, 3)$.

```
> G := ASL(2, 3);
> L := SubgroupLattice(G : Properties := true, Normalizers:= true,
>                          Centralizers:= true);
> L;
Partially ordered set of subgroup classes
----------------------------------------

[ 1]  Order 1 Length 1 C = [20] N = [20]
```

```
      Maximal Subgroups:
---
[ 2]  Order 2 Length 9 Cyclic. C = [16] N = [16]
      Maximal Subgroups: 1
[ 3]  Order 3 Length 12 Cyclic. C = [14] N = [14]
      Maximal Subgroups: 1
[ 4]  Order 3 Length 24 Cyclic. C = [10] N = [10]
      Maximal Subgroups: 1
[ 5]  Order 3 Length 4 Cyclic. C = [15] N = [18]
      Maximal Subgroups: 1
---
[ 6]  Order 4 Length 27 Cyclic. C = [6] N = [12]
      Maximal Subgroups: 2
[ 7]  Order 6 Length 12 Soluble. C = [3] N = [14]
      Maximal Subgroups: 2 5
[ 8]  Order 6 Length 36 Cyclic. C = [8] N = [8]
      Maximal Subgroups: 2 3
[ 9]  Order 9 Length 4 Elementary Abelian. C = [9] N = [18]
      Maximal Subgroups: 3 5
[10]  Order 9 Length 8 Elementary Abelian. C = [10] N = [15]
      Maximal Subgroups: 4 5
[11]  Order 9 Length 1 Elementary Abelian. C = [11] N = [20]
      Maximal Subgroups: 5
---
[12]  Order 8 Length 9 Nilpotent. C = [2] N = [16]
      Maximal Subgroups: 6
[13]  Order 18 Length 1 Soluble. C = [1] N = [20]
      Maximal Subgroups: 7 11
[14]  Order 18 Length 12 Soluble. C = [3] N = [14]
      Maximal Subgroups: 7 8 9
[15]  Order 27 Length 4 Nilpotent. C = [5] N = [18]
      Maximal Subgroups: 9 10 11
---
[16]  Order 24 Length 9 Soluble. C = [2] N = [16]
      Maximal Subgroups: 8 12
[17]  Order 36 Length 3 Soluble. C = [1] N = [19]
      Maximal Subgroups: 6 13
[18]  Order 54 Length 4 Soluble. C = [1] N = [18]
      Maximal Subgroups: 13 14 15
---
[19]  Order 72 Length 1 Soluble. C = [1] N = [20]
      Maximal Subgroups: 12 17
---
[20]  Order 216 Length 1 Soluble. C = [1] N = [20]
      Maximal Subgroups: 16 18 19
```

57.11.2.2 Operations on Subgroup Class Posets

In the following, L is the poset of subgroup classes for a group G.

`#L`

The cardinality of L, i.e., the number of conjugacy classes of subgroups of G.

`L ! i`

Create the i-th element of the poset L. The elements of L are sorted so that classes i and j of groups whose orders o_i and o_j are the products of e_i and e_j prime numbers respectively will be ordered so that i comes before j is $e_i < e_j$ or $e_i = e_j$ and $o_i < o_j$.

`L ! H`

Create the element of the poset L corresponding to the subgroup H of the group G.

`Bottom(L)`

Create the bottom of the poset L, i.e., the element of L corresponding to the trivial subgroup of G. If the poset was created with restrictions on the type of subgroups constructed, the bottom of the poset may not be the trivial subgroup.

`Top(L)`

Create the top of the poset L, i.e., the element of L corresponding to G.

`Random(L)`

Create a random element of L.

Example H57E21

We create the subgroup lattice of $A\Gamma L(1, 8)$ and locate the Fitting subgroup in the lattice.

```
> G := AGammaL(1, 8);
> L := SubgroupLattice(G);
> L;

Subgroup Lattice
----------------

[ 1]  Order 1 Length 1
      Maximal Subgroups:
---
[ 2]  Order 2 Length 7
      Maximal Subgroups: 1
[ 3]  Order 3 Length 28
      Maximal Subgroups: 1
[ 4]  Order 7 Length 8
      Maximal Subgroups: 1
---
[ 5]  Order 4 Length 7
```

```
      Maximal Subgroups: 2
[ 6]  Order 6 Length 28
      Maximal Subgroups: 2 3
[ 7]  Order 21 Length 8
      Maximal Subgroups: 3 4
---
[ 8]  Order 8 Length 1
      Maximal Subgroups: 5
[ 9]  Order 12 Length 7
      Maximal Subgroups: 3 5
---
[10]  Order 24 Length 7
      Maximal Subgroups: 6 8 9
[11]  Order 56 Length 1
      Maximal Subgroups: 4 8
---
[12]  Order 168 Length 1
      Maximal Subgroups: 7 10 11

> F := FittingSubgroup(G);
> F;
Permutation group F acting on a set of cardinality 8
Order = 8 = 2^3
    (1, 2)(3, 6)(4, 8)(5, 7)
    (1, 6)(2, 3)(4, 7)(5, 8)
    (1, 5)(2, 7)(3, 4)(6, 8)
> L!F;
8
```

We now construct a chain from the bottom to the top of the lattice.

```
> H := Bottom(L);
> Chain := [H];
> while H ne Top(L) do
>     H := Representative(MinimalOvergroups(H));
>     Chain := Append(Chain, H);
> end while;
> Chain;
[ 1, 2, 5, 8, 10, 12 ]
```

57.11.2.3 Operations on Poset Elements

In the following, L is the poset of subgroups for a group G. Elements of L are identified with the integers $[1..\#L]$.

`IntegerRing() ! e`

The integer corresponding to poset element e.

`e eq f`

Returns `true` if and only if poset elements e and f are equal.

`e ge f`

Returns `true` if and only if poset element e contains poset element f.

`e ge f`

Returns `true` if and only if poset element e strictly contains poset element f.

`e le f`

`e subset f`

Returns `true` if and only if poset element e is contained in poset element f.

`e lt f`

Returns `true` if and only if poset element e is strictly contained in poset element f.

57.11.2.4 Class Information from a Conjugacy Class Poset

In the following, L is the poset of subgroups for a group G. Elements of L are identified with the integers $[1..\#L]$.

`Group(e)`

The subgroup of G that is the chosen class representative corresponding to the element e of the poset L.

`Centraliser(e, f)`

`Centralizer(e, f)`

Given poset elements e and f, return the poset element that corresponds to the class of subgroups that contains the centralizers of the subgroups of class f (taken in a subgroup of class e). If no subgroup of class f lies in class e, the construction fails.

`Normaliser(e, f)`

`Normalizer(e, f)`

Given poset elements e and f, return the poset element that corresponds to the class of subgroups that contain the normalizers of the subgroups of class f (taken in a subgroup of class e). If no subgroup of class f lies in class e, the construction fails.

`Length(e)`

The number of subgroups in the class corresponding to e.

`Order(e)`

The order of the subgroup of G corresponding to e.

`MaximalSubgroups(e)`

The maximal subgroups of e, returned as a set of poset elements.

`MinimalOvergroups(e)`

The minimal overgroups of e, returned as a set of poset elements.

`NumberOfInclusions(e, f)`

The number of elements of the conjugacy class of subgroups e that lie in a fixed representative of the conjugacy class of subgroups f.

57.12 Cohomology

In the following description, G is a group in the category `GrpPerm`, p is a prime number, and K is the finite field of order p. Further, F is a finitely presented group having the same number of generators as G, and is such that its relations are satisfied by the corresponding generators of G. In other words, the mapping taking the i-th generator of F to the i-th generator of G must be an epimorphism. Usually this mapping will be an isomorphism, although this is not mandatory.

`pMultiplicator(G, p)`

Given the group G and a prime p, return the invariant factors of the p-part of the Schur multiplicator of G.

`pCover(G, F, p)`

Given the group G and the finitely presented group F such that G is an epimorphic image of G in the sense described above, return a presentation for the p-cover of G, constructed as an extension of the p-multiplier by F.

`CohomologicalDimension(G, M, i)`

Given the group G, the $K[G]$-module M and an integer i (equal to 1 or 2), return the dimension of the i-th cohomology group of G acting on M.

`ExtensionProcess(G, M, F)`

Create an extension process for the group G by the module M.

`Extension(P, Q)`

`#NextExtension(P)`

Return the next extension of G as defined by the process P.
Assume that F is isomorphic to the permutation group G, and that we wish to determine presentations for one or more extensions of the K-module M by F, where K is the field of p elements. We first create an extension process using `ExtensionProcess(G, M, F)`. The possible extensions of M by G are in one-one correspondence with the elements of the second cohomology group $H^2(G, M)$ of G acting on M. Let $b_1, \ldots, b_l$ be a basis of $H^2(G, M)$. A general element of $H^2(G, M)$ therefore has the form $a_1b_1 + \cdots + a_lb_l$ and so can be defined by a sequence Q of l integers $[a_1, \ldots, a_l]$. Now, to construct the corresponding extension of M by G we call the function `Extension(P, Q)`. The required extension is returned as a finitely presented group. If all the extensions are required then they may be obtained successively by making p^l calls to the function `NextExtension`.

`SplitExtension(G, M, F)`

The split extension of the module M by the group G.

57.13 Characters and Representations

A set of functions are provided for computing with the characters and representations of a group. A full account of the character functions may be found in Chapter 91. Full details of the functions for constructing and analyzing representations may be found in Chapter 89. For the reader's convenience we include here a description of the basic functions for creating characters and representations.

Some functions described in this section may be missing or may have slightly different calling sequences for some categories of groups. For a complete description of the features available for a special category of groups, we refer to the chapter devoted to that category.

57.13.1 Character Theory

`CharacterDegrees(G)`

Given a finite pc-group G, return the sequence $[\langle d_1, c_1\rangle, langled_2, c_2\rangle, \ldots]$, where c_i is the number of irreducible characters of G having degree d_i. For details of the algorithm see Conlon [Con90b].

`CharacterTable(G)`

Construct the table of irreducible characters for the group G.

`PermutationCharacter(G)`

Given a group G represented as a permutation group, construct the character of G afforded by the defining permutation representation of G.

`PermutationCharacter(G, H)`

Given a group G and some subgroup H of G, construct the ordinary character of G afforded by the permutation representation of G given by the action of G on the coset space of the subgroup H in G.

57.13.2 Representation Theory

We describe the main functions for creating $K[G]$-modules for finite groups. The machinery for working with these modules is described in Chapter 89.

`GModule(G, S)`

Let G be a group defined on r generators and let S be a subalgebra of the matrix algebra $M_n(R)$, also defined by r non-singular matrices. It is assumed that the mapping from G to S defined by $\phi : G.i \mapsto S.i$, for $i = 1, \ldots, r$, extends to a group homomorphism. Let M be the natural module for the matrix algebra S. The function `GModule` gives M the structure of an $S[G]$-module, where the action of the i-th generator of G on M is given by the i-th generator of S.

`GModule(G, A, B)`

Given a finite group G, a normal subgroup A of G and a normal subgroup B of A such that the section A/B is elementary abelian of order p^n, create the $K[G]$-module M corresponding to the action of G on A/B, where K is the field $\mathbf{F}_p$. If B is trivial, it may be omitted. The function returns:

(a) the module M; and,

(b) the homomorphism $\phi : A/B \to M$.

`PermutationModule(G, H, R)`

Given a finite group G and a ring R, create the $R[G]$-module for G corresponding to the permutation action of G on the cosets of H.

`PermutationModule(G, R)`

Given a finite permutation group G and a ring R, create the natural permutation module for G over R.

Example H57E22

The permutation module for the group M_{10} over $GF(2)$ may be created as follows:

```
> m10 := PermutationGroup< 10 |  (1, 3, 9, 10, 2, 8, 7, 6, 4, 5),
>                                (1, 7)(2, 4, 3, 6, 8, 10, 9, 5) >;
> p := PermutationModule(m10, GF(2));
> p : Maximal;

GModule p of dimension 10 over GF(2)
Generators of acting algebra:
```

```
[0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 1 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 1]
[0 1 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 1 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0]
[1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0]
```

Example H57E23

The group G defined below is the split extension of an elementary abelian group E of order 16 by $Alt(6)$. After setting up the group, we construct the module M for G corresponding to its action on E.

```
> G := PermutationGroup< 16 |
>         (1, 15, 7, 5, 12)(2, 9, 13, 14, 8)(3, 6, 10, 11, 4),
>         (1, 4, 5)(2, 8, 10)(3, 12, 15)(6, 13, 11)(7, 9, 14),
>         (1, 16)(2, 3)(4, 5)(6, 7)(8, 9)(10, 11)(12, 13)(14, 15) >;
> CS := ChiefSeries(G);
> [ Order(H) : H in CS ];
[ 5760, 16, 1 ]
> M := GModule(G, CS[2]);
> M:Maximal;

GModule M of dimension 4 over GF(2)
Generators of acting algebra:

[0 1 0 0]
[0 1 1 0]
[0 0 1 1]
[1 0 0 1]

[0 0 1 0]
[0 0 0 1]
```

```
[1 0 1 0]
[0 1 0 1]

[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

57.14 Databases of Groups

MAGMA contains the following databases of groups:

Small Groups: Contains all groups of order up to 1000, excluding orders 512 and 768.

Perfect Groups: This database contains all perfect groups up to order 50000, and many classes of perfect groups up to order one million. Each group is defined by means of a finite presentation. Further information is also provided which allows the construction of permutation representations.

Rational Maximal Matrix Groups: Contains rational maximal finite matrix groups and their invariant forms, for small dimensions (up to 31 at V2.9 and above). Each entry can be accessed either as a matrix group or as a lattice.

Quaternionic Matrix Groups: A database of the finite absolutely irreducible subgroups of $\mathrm{GL}_n(\mathcal{D})$ where $\mathcal{D}$ is a definite quaternion algebra whose centre has degree d over $\mathbf{Q}$ and $nd \leq 10$. Each entry can be accessed either as a matrix group or as a lattice.

Transitive Permutation Groups: MAGMA has a database containing all transitive permutation groups having degree up to 22.

Primitive Permutation Groups: MAGMA has a database containing all primitive permutation groups having degree up to 50.

For a description of these databases, we refer to Chapter 66.

57.15 Bibliography

[**Con90**] S. B. Conlon. Computing modular and projective character degrees of soluble groups. *J. Symbolic Comp.*, 9:551–570, 1990.

[**LGM02**] C. R. Leedham-Green and Scott H. Murray. Variants of product replacement. *Contemp. Math.*, 298:97–104, 2002.

[**Lo98**] Eddie H. Lo. Finding intersections and normalizers in finitely generated nilpotent groups. *J. Symbolic Comput.*, 25(1):45–59, 1998.

58 PERMUTATION GROUPS

Chapter 58
PERMUTATION GROUPS

58.1 Introduction

58.1.1 Terminology

A permutation group G is a group of bijections $X \to X$, for some set X. The group G is said to *act* on X and the elements of G are called permutations (of the set X). A given permutation group G may have actions on sets other than the one on which it is defined. Thus, any set upon which G has a legitimate action will be called a G-set. The set X is called the *natural G-set* for the group G, and the action of G on X is called the *natural action* of G. Note that the group G also has a natural induced action on the G-closure of any *derived set* of X (see Section 14.8.1). MAGMA expects the G-set X to be of finite cardinality n. Usually, X will be $\{1, 2, \ldots, n\}$, but, as we shall see below, X may be a set of strings, or any other legitimate MAGMA set.

The elements of a G-set are called *points*. Let Y be a G-set for G. The (possibly empty) subset of Y whose points are fixed by every permutation of G, is called the *fixed-point set* for G, while the subset of Y consisting of points moved by some permutation of G is called the *support* of G. Similarly, for an element g of G the *fixed-point set* and the *support* of g are, respectively, the subsets of Y consisting of the points fixed and moved by g. The *degree* of G is defined to be the cardinality of the natural G-set of G; whereas the *degree* of an element g of G is defined to be the cardinality of the support of g, i.e. the number of points moved by g.

Permutation groups in MAGMAare limited to degree less than 2^{30}.

58.1.2 The Category of Permutation Groups

The family of all permutation groups of finite degree forms a category. The objects are the permutation groups and the morphisms are group homomorphisms. The MAGMA designation for this category of permutation groups is `GrpPerm`.

58.1.3 The Construction of a Permutation Group

Every permutation group acting on a set X is created as a subgroup of the symmetric group $\mathrm{Sym}(X)$. Thus, the construction of a general permutation group is a two-step process:

(i) The appropriate symmetric group, $\mathrm{Sym}(X)$, is constructed;

(ii) The required group G is then defined as a subgroup of $\mathrm{Sym}(X)$.

For convenience, a constructor `PermutationGroup< ... >`, which combines these two steps, is provided.

58.2 Creation of a Permutation Group

58.2.1 Construction of the Symmetric Group

`Sym(n)`

`SymmetricGroup(n)`

Given an integer $n \geq 1$, create the generic permutation group acting on the natural G-set $\Omega = \{1, 2, \ldots, n\}$, i.e. the symmetric group $\mathrm{Sym}(\Omega)$. Initially, only a structure table is created for $\mathrm{Sym}(n)$, so that, in particular, generators are not defined. This function is normally used to provide a context for the creation of elements and subgroups of $\mathrm{Sym}(n)$. If structural computation is attempted with the group created by `Sym(n)`, then generators will be created dynamically.

`Sym(X)`

`SymmetricGroup(X)`

Given a finite set X of cardinality $n \geq 1$, create the generic group G of permutations of X – the symmetric group $\mathrm{Sym}(X)$. Initially, only a structure table is created for $\mathrm{Sym}(X)$, so that, in particular, generators are not defined. This function is normally used to provide a context for the creation of elements and subgroups of $\mathrm{Sym}(X)$. If structural computation is attempted with the group created by `Sym(X)`, then generators will be created dynamically. Although the group G is defined on the set X, G is represented internally as a group of permutations of the set $\Omega = \{1, 2, \ldots, n\}$. Translation between X and Ω is done at input/output time. The precise representation can be found by using the `Labelling` function. If X is an indexed set then the indexing of elements of X determines the correspondence.

`StandardGroup(G)`

Return a group H isomorphic to G, but acting on the standard set $\{ 1, \ldots, n\}$. This function is useful when the natural G-set for G is not the standard set. If the natural G-set for G is the standard set, G is returned. The isomorphism from G to H is also returned.

Example H58E1

We define the symmetric group on the set of strings { "*a*" "*b*" "*c*" "*d*" }:

```
> S4 := Sym({ "a", "b", "c", "d" });
> S4;
Symmetric group S4 acting on a set of cardinality 4
Order = 24 = 2^3 * 3
> GSet(S4);
GSet{@ c, b, a, d @}
```

We define the symmetric group of degree 10 acting on the set $\{0, 1, \ldots, 9\}$.

```
> G := Sym({ 0..9 });
```

```
> G;
Symmetric group G acting on a set of cardinality 10
Order = 3628800 = 2^8 * 3^4 * 5^2 * 7
> GSet(G);
GSet{@ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 @}
```

58.2.2 Construction of a Permutation

Throughout this subsection we shall assume that the permutation group G has natural G-set X.

`elt< G | L >`

Given a permutation group G defined as acting on the set $X = \{x_1, \ldots, x_n\}$ of cardinality $n \geq 1$, and a list L of distinct elements $a_1, a_2, \ldots, a_n$ of X, construct the element g of G defined by $x_i \to a_i$, for $i = 1, \ldots, n$. Unless G is known to be the generic permutation group of degree n, the permutation will be tested for membership of G, and if g is not an element of G, the function will fail. If g does lie in G, g will have G as its parent. Since the membership test may involve constructing a base and strong generating set for G, this constructor may occasionally be very costly. Hence, a permutation g should be defined as an element of a subgroup of the generic group only when membership of G is required by subsequent operations involving g.

`G ! Q`

Given a permutation group G defined as acting on the set $X = \{x_1, \ldots, x_n\}$ of cardinality $n \geq 1$, and a sequence $Q = [a_1, a_2, \ldots, a_n]$ of distinct elements of X, construct the permutation g of X defined by $x_i \to a_i$, for $i = 1, \ldots, n$. This permutation will have G as its parent structure. As in the case of the `elt`-constructor, the operation will fail if g is not an element of G and the same observations concerning the cost of membership testing apply.

`G ! (...)(...)...(...)`

Given a permutation group G defined as acting on the set $X = \{x_1, x_2, \ldots, x_n\}$, construct the permutation g corresponding to the given product of cycles. Adjacent letters must be separated by commas. Further, *cycles of length one must be omitted*. The coercion operator ! may be omitted only within the context of the standard constructors `sub<>`, `ncl<>` and `quo<>`. Once the permutation g has been constructed, it will be tested for membership in G. If it is not a member, the construction fails.

`G ! \(...)(...)...(...)`

Given a permutation group G defined as acting on the set $X = \{1..n\}$, construct the permutation g corresponding to the given product of literal cycles of integers. Adjacent integers must be separated by commas. Once the permutation g has been constructed, it will be tested for membership in G. If it is not a member, the construction fails. This construction is strongly recommended when creating large permutations to avoid overhead in constructing unnecessarily large parse trees by MAGMA.

`G ! Q`

Given a permutation group G defined as acting on the set $X = \{1..n\}$, construct the permutation g corresponding to the given product of cycles. The indexed sets in Q must be disjoint subsets of X, which are interpreted as the disjoint cycles of the permutation being constructed. Cycles of length 1 may be omitted, but do not have to be omitted. Once the permutation g has been constructed, it will be tested for membership in G. If it is not a member, the construction fails. Note that the `Cycle` function produces results suitable for use as members of Q.

`ElementToSequence(g)`

`Eltseq(g)`

The sequence Q of images of the G-set of g. In particular, it has the property that `Parent(g)!Eltseq(g) eq g`.

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity permutation in the permutation group G.

Example H58E2

The three different constructions are illustrated by the following code, which assigns to each of the variables x, y and z the permutation (1)(2,3)(4,5,6).

```
> S6 := Sym(6);
> x := elt<S6 | 1,3,2,5,6,4>;
> x;
(2, 3)(4, 5, 6)
> y := S6![1,3,2,5,6,4];
> y;
(2, 3)(4, 5, 6)
> z := S6!(2,3)(4,5,6);
> z;
(2, 3)(4, 5, 6)
> S6!1;
Id(S6)
```

58.2.3 Construction of a General Permutation Group

`PermutationGroup< X | L >`

Suppose that the cardinality of the set X is n. Construct the permutation group G acting on the set X generated by the permutations defined by the list L. A term of the list L must be an object of one of the following types:

(a) A sequence of n elements of X defining a permutation of X (note that this is only well-defined when X is an indexed set);

(b) A set or sequence of sequences of type (a);

(c) An element of $\mathrm{Sym}(X)$;

(d) A set or sequence of elements of $\mathrm{Sym}(X)$;

(e) A subgroup of $\mathrm{Sym}(X)$;

(f) A set or sequence of subgroups of $\mathrm{Sym}(X)$.

Each element or group specified by the list must belong to the *same* generic permutation group. The group G will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of G consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list.

The `PermutationGroup` constructor is shorthand for the two statements:

```
SX := Sym(X); G := sub< SX | L >;
```

where `sub< ... >` is the subgroup constructor described in the next subsection.

`PermutationGroup< n | L >`

Construct the permutation group G acting on the set $X = \{1, 2, \ldots, n\}$ generated by the permutations defined by the list L. The possibilities for the terms of the list L are the same as for the constructor `PermutationGroup< X | L >`.

Example H58E3

The Hessian group generated by the permutations $(1,2,4)(5,6,8)(3,9,7)$ and $(4,5,6)(7,9,8)$ may be created by the statement:

```
> H := PermutationGroup< 9 | (1,2,4)(5,6,8)(3,9,7), (4,5,6)(7,9,8) >;
> H;
Permutation group H acting on a set of cardinality 9
       (1, 2, 4)(3, 9, 7)(5, 6, 8)
       (4, 5, 6)(7, 9, 8)
```

58.3 Elementary Properties of a Group

58.3.1 Accessing Group Information

The functions in this group provide access to basic information stored for a permutation group G.

`G . i`

The i-th defining generator for G. A negative subscript indicates that the inverse of the generator is to be created. The identity element of G will be created by $G.0$.

`Degree(G)`

The degree of the permutation group G.

`Generators(G)`

A set of elements of G that generate G.

`GeneratorsSequence(G)`

The sequence of elements used to define the group G. Any occurrences of the identity element or any repetitions of a generator, as removed by `Generators(G)`, are retained in this sequence. This function has the same effect as the expression `[G.i : i in [1..Ngens(G)]]` .

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for G.

`FewGenerators(G)`

A typically short sequence of random elements generating the group. Especially when groups are generated as subgroups, the result of `FewGenerators` is a much shorter sequence than returned by `GeneratorsSequence`.

`Generic(G)`

The generic group containing G, i.e. the symmetric group in which G is naturally embedded.

`Parent(g)`

The parent group G for the permutation g.

`GSet(G)`

The natural G-set for the permutation group G.

Example H58E4

Consider the group G of order 648 generated by the permutations (1,6,7)(2,5,8,3,4,9)(11,12) and (1,3)(4,9,12)(5,8,10,6,7,11).

```
> G := PermutationGroup< 12 | (1,6,7)(2,5,8,3,4,9)(11,12),
>                             (1,3)(4,9,12)(5,8,10,6,7,11) >;
> G;
Permutation group G acting on a set of cardinality 12
    (1, 6, 7)(2, 5, 8, 3, 4, 9)(11, 12)
    (1, 3)(4, 9, 12)(5, 8, 10, 6, 7, 11)
> G.1;
(1, 6, 7)(2, 5, 8, 3, 4, 9)(11, 12)
> G.1*G.2;
(1, 7, 3, 9, 2, 8)(4, 12, 5, 10, 6, 11)
> Degree(G);
12
> GSet(G);
GSet{@ 1 .. 12 @}
> Generic(G);
Symmetric group acting on a set of cardinality 12
Order = 479001600 = 2^10 * 3^5 * 5^2 * 7 * 11
> Generators(G);
{
    (1, 6, 7)(2, 5, 8, 3, 4, 9)(11, 12),
    (1, 3)(4, 9, 12)(5, 8, 10, 6, 7, 11)
}
> Ngens(G);
2
> x := G ! (1,6,7)(2,5,8,3,4,9)(11,12);
> x;
(1, 6, 7)(2, 5, 8, 3, 4, 9)(11, 12)
> Parent(x);
Permutation group G acting on a set of cardinality 12
Order = 648 = 2^3 * 3^4
    (1, 6, 7)(2, 5, 8, 3, 4, 9)(11, 12)
    (1, 3)(4, 9, 12)(5, 8, 10, 6, 7, 11)
]
```

58.3.2 Group Order

Unless the order is already known, each of the functions in this family will create a base and strong generating set for the group if one does not already exist.

`Order(G)`
`#G`

The order of the group G as an integer. If the order is not currently known, a base and strong generating set will be constructed for G.

`FactoredOrder(G)`

The order of the group G returned as a factored integer. The factorization is returned in the form of a sequence Q which is defined as follows: If $\#G = p_1^{e_1} \dots p_n^{e_n}, \quad e_i \neq 0$, then Q will be the integer sequence $[< p_1, e_1 >, \dots, < p_n, e_n >]$. If the order of G is not known, it will be computed.

58.3.3 Abstract Properties of a Group

`IsAbelian(G)`

Returns `true` if the group G is abelian, `false` otherwise.

`IsCyclic(G)`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group G is elementary abelian, `false` otherwise.

`IsSpecial(G)`

Given a p-group G, return `true` if G is special, `false` otherwise.

`IsExtraSpecial(G)`

Given a group G is a p-group G, return `true` if G is extra-special, `false` otherwise.

`IsNilpotent(G)`

Returns `true` if the group G is nilpotent, `false` otherwise.

`IsSoluble(G)`
`IsSolvable(G)`

Returns `true` if the group G is soluble, `false` otherwise. Uses the algorithm of Sims [Sim90].

`IsPerfect(G)`

Returns `true` if the group G is perfect, `false` otherwise.

`IsSimple(G)`

Returns `true` if the group G is simple, `false` otherwise.

`IsWreathProduct(G)`

Returns `true` if the group G is isomorphic to a wreath product $A \wr B$, where B is transitive, and `false` otherwise. If true, then three subgroups of G, call them A, B, C, are also returned. In this case we have G isomorphic to `WreathProduct(A, CosetImage(B, C))`.

Example H58E5

We determine the orders of those subgroups of the Mathieu group M_{24} which are perfect but not simple. We use the function `PerfectSubgroups` which returns a representative from each conjugacy class of perfect subgroups.

```
> load m24;
Loading "/home/magma/libs/pergps/m24"
M24 - Mathieu group on 24 letters - degree 24
Order 244 823 040 = 2^10 * 3^3 * 5 * 7 * 11 * 23;  Base 1,2,3,4,5,6,7
Group: G
> time S := PerfectSubgroups(G);
Time: 29.460
> [ Order(H) : R in S | not IsSimple(H) where H := R`subgroup ];
[ 120, 120, 120, 180, 180, 240, 240, 336, 336, 336, 336, 504, 720, 1008, 1080,
  960, 960, 960, 1344, 1344, 1344, 1920, 2688, 2688, 2688, 2688, 2688, 2880,
  3840, 3840, 5760, 10752, 11520, 11520, 40320, 21504, 21504, 32256, 64512,
  69120, 322560 ]
```

58.4 Homomorphisms

Homomorphisms are a central concept in group theory, and MAGMA provides extensive facilities for group homomorphisms. Many useful homomorphisms are returned by constructors and intrinsic functions. Examples of these are the `quo` constructor, the `sub` constructor and intrinsic functions such as `OrbitAction`, `BlocksAction`, `FPGroup` and `RadicalQuotient`, which are described in more detail elsewhere in this chapter. In this section we describe how the user may create their own homomorphisms with domain a permutation group.

`hom< G -> H | L >`

Given the permutation group G, construct the homomorphism $f : G \to H$ given by the generator images in L. H must be a group. The clause L may be any one of the following types:

(a) A list of elements of H, giving images of the generators of G;

(b) A list of pairs, where the first in the pair is an element of G and the second its image in H;

(c) A sequence of elements of H, as in (a);

(d) A set or sequence of pairs, as in (b);

Each image element specified by the list must belong to the *same* group H. In the cases where pairs are given the given elements of G must generate G.

`Domain(f)`

The domain of the homomorphism f.

`Codomain(f)`

The codomain of the homomorphism f.

`Image(f)`

The image or range of the homomorphism f. This will be a subgroup of the codomain of f. The algorithm computes the image and kernel simultaneously (see [LGPS91]).

`Kernel(f)`

The kernel of the homomorphism f. This will be a normal subgroup of the domain of f. The algorithm computes the image and kernel simultaneously (see [LGPS91]).

`IsHomomorphism(G, H, Q)`

Return the value `true` if the sequence Q defines a homomorphism from the group G to the group H. The sequence Q must have length `Ngens(G)` and must contain elements of H. The i-th element of Q is interpreted as the image of the i-th generator of G and the function decides if these images extend to a homomorphism. If so, the homomorphism is also returned. The algorithm employed is described in [LGPS91].

Example H58E6

Consider the group G of order 648 generated by the permutations (1,6,7)(2,5,8,3,4,9)(11,12) and (1,3)(4,9,12)(5,8,10,6,7,11). We construct a permutation representation of G of degree 8 by considering the conjugation action of G on one of its elements. We then construct the preimage of a normal subgroup of the image.

```
> G := PermutationGroup< 12 | (1,6,7)(2,5,8,3,4,9)(11,12),
>                             (1,3)(4,9,12)(5,8,10,6,7,11) >;
> #G;
648
> x := G ! (1, 2, 3)(7, 8, 9)(10, 11, 12);
> x_class := {@ x ^ y : y in G @};
> #x_class;
8
> S := SymmetricGroup(8);
> images := [S![Index(x_class, x_class[i]^(G.j)):i in [1..8]] :j in [1..2]];
> f := hom< G -> S | images>;
```

The map f is the homomorphism of G onto the group induced by the action of the element x. We computer the images of some elements and then find the image and kernel of f.

```
> (G.1*G.-2) @ f;
(2, 5, 7)(3, 8, 6)
> ((G.1) @ f) * ((G.2) @ f) ^ -1;
(2, 5, 7)(3, 8, 6)
> H := Image(f);
> H;
Permutation group acting on a set of cardinality 8
Order = 24 = 2^3 * 3
  (1, 2, 3, 4, 6, 5)(7, 8)
  (1, 2, 8, 4, 6, 7)(3, 5)
> Kernel(f);
Permutation group acting on a set of cardinality 12
Order = 27 = 3^3
  (1, 2, 3)(4, 6, 5)(7, 8, 9)(10, 12, 11)
  (4, 5, 6)(7, 9, 8)
  (7, 9, 8)(10, 11, 12)
```

We now find the preimage of $O_2(H)$ as a subgroup of G.

```
> pCore(H, 2) @@ f;
Permutation group acting on a set of cardinality 12
Order = 216 = 2^3 * 3^3
  (4, 5, 6)(7, 9, 8)
  (1, 2, 3)(4, 6, 5)(7, 8, 9)(10, 12, 11)
  (1, 4, 2, 5, 3, 6)(7, 12, 9, 11, 8, 10)
  (1, 10, 3, 11, 2, 12)(4, 9, 5, 8, 6, 7)
  (2, 3)(4, 5)(8, 9)(11, 12)
  (7, 9, 8)(10, 11, 12)
```

58.5 Building Permutation Groups

Examples of permutation groups are routinely constructed by taking one or more standard groups and applying some extension procedure to construct a group having the given groups as subgroups or quotient groups. In the first subsection we describe functions which construct some well-known groups and in the following subsection we give functions for constructing direct and wreath products.

58.5.1 Some Standard Permutation Groups

A number of functions are provided which construct various standard groups. The effect of these functions is to construct the group on some standard set of generating permutations.

`AbelianGroup(GrpPerm, Q)`

Construct the abelian group defined by the sequence $Q = [n_1, \ldots, n_r]$ of positive integers. The function constructs the direct product of cyclic groups

$$Z(n_1) \times Z(n_2) \times \cdots \times Z(n_r).$$

`AlternatingGroup(GrpPerm, n)`

`AlternatingGroup(n)`

`Alt(n)`

Construct the alternating group of degree n on generators $(3, 4, \ldots, n)$ and $(1, 2, 3)$, if n is odd, or $(1, 2)(3, 4, \ldots, n)$ and $(1, 2, 3)$, if n is even.

`CyclicGroup(GrpPerm, n)`

`CyclicGroup(n)`

Construct the cyclic group of order n with generator $(1, 2, \ldots, n)$.

`DihedralGroup(GrpPerm, n)`

`DihedralGroup(n)`

Construct the dihedral group of degree n and order $2 * n$ on generators $(1, 2, \ldots, n)$ and $(1, n)(2, n-1)\cdots$.

`Sym(GrpPerm, n)`

`SymmetricGroup(GrpPerm, n)`

`Sym(n)`

`SymmetricGroup(n)`

Construct the symmetric group of degree n on generators $(1, 2, \ldots, n)$ and $(1, 2)$.

`ExtraSpecialGroup(GrpPerm, p, n : parameters)`

`ExtraSpecialGroup(p, n : parameters)`

Given a small prime p and a small positive integer n, construct an extra-special group G of order p^{2n+1} in the category `GrpPerm`. The isomorphism type of G can be selected using the parameter `Type`.

`Type` MonStgElt *Default :* "+"

Possible values for this parameter are "+" (default) and "−".

If `Type` is set to "+", the function returns for $p = 2$ the central product of n copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p.

If `Type` is set to "−", the function returns for $p = 2$ the central product of a quaternion group of order 8 and $n - 1$ copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p^2.

`YoungSubgroup(L)`

`Full` RngIntElt *Default :* `false`

Given a sequence L of positive integers, compute the Young subgroup parameterized by L, i.e., the direct product of the symmetric groups on L_i points. If the optional parameter `Full` is given, construct the group as a subgroup of the symmetric group on `Full` elements.

Example H58E7

(1) The abelian group $Z_2 \times Z_2 \times Z_4$:

```
> A := AbelianGroup(GrpPerm, [2, 2, 4] );
> A;
Permutation group A acting on a set of cardinality 8
Order = 16 = 2^4
    (1, 2)
    (3, 4)
    (5, 6, 7, 8)
```

(2) The alternating group of degree 12:

```
> A12 := AlternatingGroup(GrpPerm, 12);
> A12;
Permutation group A12 acting on a set of cardinality 12
Order = 239500800 = 2^9 * 3^5 * 5^2 * 7 * 11
    (1, 2)(3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
    (1, 2, 3)
```

(3) The cyclic group Z_{24}:

```
> Z24 := CyclicGroup(GrpPerm, 24);
> Z24;
Permutation group Z24 on a set of cardinality 24
```

```
Order = 24 = 2^3 * 3
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
        16, 17, 18, 19, 20, 21, 22, 23, 24)
```

(4) The dihedral group of order 24:

```
> D12 := DihedralGroup(GrpPerm, 12);
> D12;
Permutation group D12 acting on a set of cardinality 12
Order = 24 = 2^3 * 3
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
    (1, 12)(2, 11)(3, 10)(4, 9)(5, 8)(6, 7)
```

(5) The symmetric group of degree 8:

```
> S8 := SymmetricGroup(GrpPerm, 8);
> S8;
Symmetric group S8 acting on a set of cardinality 8
Order = 40320 = 2^7 * 3^2 * 5 * 7
```

58.5.2 Direct Products and Wreath Products

`DirectProduct(G, H)`

Given two permutation groups G and H, construct the direct product D of G and H as an intransitive group having degree equal to the sum of the degrees of G and H. In addition, the sequences I of inclusions and P of projections are returned, satisfying $I[i] : K_i \to D(K_i)$ and $P[i] : D \to K_i$ (where $K_1 = G, K_2 = H$ and $D(K)$ is the group K represented naturally as a subgroup of D).

`DirectProduct(Q)`

Given a sequence Q of n permutation groups, construct the direct product $Q[1] \times Q[2] \times \ldots \times Q[n]$ as an intransitive group of degree equal to the sum of the degrees of the groups $Q[i]$, $(i = 1, \ldots, n)$. In addition, the sequences I of inclusion and P of projections are returned, satisfying $I[i] : Q[i] \to D(Q[i])$ and $P[i] : D \to Q[i]$ (where $D(K)$ is the group K represented naturally as a subgroup of D).

`PrimitiveWreathProduct(G, H)`

Given permutation groups G and H, construct the wreath product $G \wr H$ of G and H, where $G \wr H$ has product action.

`PrimitiveWreathProduct(Q)`

Given a sequence Q of n permutation groups, construct the iterated wreath product $T = (\ldots (Q[1] \wr Q[2]) \wr \ldots \wr Q[n])$, where T has product action.

`WreathProduct(G, H)`

Given permutation groups G and H, construct the wreath product $W = G \wr H$ of G and H, where $G \wr H$ has imprimitive action. The function also returns the sequence of `Degree(`H`)` inclusions of G into W, the inclusion of H into W and the projection of W onto H.

`WreathProduct(Q)`

Given a sequence Q of n permutation groups, construct the iterated wreath product $W = (\ldots(Q[1] \wr Q[2]) \wr \ldots \wr Q[n])$, where W has imprimitive action.

`WreathProduct(B)`

Given a block system B of some permutation group G, compute the wreath-product corresponding to B.

`WreathProduct(G, B)`

Compute the smallest wreath product W to the block system B of G such that $G \subseteq W$. Also return the complement as a subgroup of W. The third parameter is a subgroup which is isomorphic to the action within a block.

Example H58E8

We define G to be the symmetric group of degree 4 and H to be the dihedral group of order 8. We then proceed to form the direct, primitive-wreath and wreath products of G and H.

```
> G := SymmetricGroup(GrpPerm, 4);
> H := DihedralGroup(GrpPerm, 3);
> D := DirectProduct(G, H);
> D;
Permutation group D acting on a set of cardinality 7
Order = 144 = 2^4 * 3^2
    (1, 2, 3, 4)
    (1, 2)
    (5, 6, 7)
    (5, 6)
> T := PrimitiveWreathProduct(G, H);
> T;
Permutation group T acting on a set of cardinality 64
Order = 82944 = 2^10 * 3^4
    (2, 5, 17)(3, 9, 33)(4, 13, 49)(6, 21, 18)(7, 25, 34)(8, 29, 50)
       (10, 37, 19) (11, 41, 35)(12, 45, 51) (14, 53, 20)(15, 57, 36)
       (16, 61, 52)(23, 26, 38) (24, 30, 54)(27, 42, 39)(28, 46, 55)
       (31, 58, 40) (32, 62, 56)(44, 47, 59)(48, 63, 60)
    (2, 5)(3, 9)(4, 13)(7, 10)(8, 14)(12, 15)(18, 21)(19 , 25)(20, 29)
       (23, 26)(24, 30)(28, 31)(34, 37)(35 , 41)(36, 45)(39, 42)(40, 46)
       (44, 47)(50, 53)(51 , 57)(52, 61)(55, 58)(56, 62)(60, 63)
    (1, 2, 3, 4)(5, 6, 7, 8)(9, 10, 11, 12)(13, 14, 15, 16)(17, 18, 19, 20)
       (21, 22, 23, 24)(25, 26, 27, 28)(29, 30, 31, 32)(33, 34, 35, 36)
```

```
       (37, 38, 39, 40)(41, 42, 43, 44)(45, 46, 47, 48)(49, 50, 51, 52)
       (53, 54, 55, 56)(57, 58, 59, 60)(61, 62, 63, 64)
    (1, 2)(5, 6)(9, 10)(13, 14)(17, 18)(21, 22)(25, 26)( 29, 30)(33, 34)
       (37, 38)(41, 42)(45, 46)(49, 50)( 53, 54)(57, 58)(61, 62)
> W := WreathProduct(G, H);
> W;
Permutation group W acting on a set of cardinality 12
Order = 82944 = 2^10 * 3^4
    (1, 5, 9)(2, 6, 10)(3, 7, 11)(4, 8, 12)
    (1, 5)(2, 6)(3, 7)(4, 8)
    (1, 2, 3, 4)
    (1, 2)
```

58.6 Permutations

58.6.1 Coercion

`G ! g`

Given a subgroup G of $\mathrm{Sym}(X)$ and a permutation g belonging to $\mathrm{Sym}(X)$ that is contained in G, embed g in G. Thus, this operator changes the parent of g to be G.

`G !! H`

Given a group H whose natural G-set X is a subset of the natural G-set Y for the group G, embed H as a subgroup of G. The operator fails if the image of H in $\mathrm{Sym}(Y)$ is not a subgroup of G.

58.6.2 Arithmetic with Permutations

`g * h`

Product of permutation g and permutation h, where g and h belong to the same generic group U. If g and h both belong to the same proper subgroup G of U, then the result will be returned as an element of G; if g and h belong to subgroups H and K of a subgroup G of U, then the product is returned as an element of G. Otherwise, the product is returned as an element of U.

`g ^ n`

The n-th power of the permutation g, where n is a positive, negative or zero integer.

`g / h`

Product of the permutation g by the inverse of the permutation h, i.e. the element $g * h^{-1}$. Here g and h must belong to the same generic group U. The rules for determining the parent group of g/h are the same as for $g * h$.

`g ^ h`

Conjugate of the permutation g by the permutation h, i.e. the element $h^{-1} * g * h$. Here g and h must belong to the same generic group U. The rules for determining the parent group of g^h are the same as for $g * h$.

`(g, h)`

Commutator of the permutations g and h, i.e. the element $g^{-1} * h^{-1} * g * h$. Here g and h must belong to the same generic group U. The rules for determining the parent group of (g, h) are the same as those for $g * h$.

`(g`$_1$`, ..., g`$_r$`)`

Given r permutations $g_1, \ldots, g_r$ belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

58.6.3 Properties of Permutations

`CycleStructure(g)`

Given a permutation g belonging to a group of degree n, return the partition of n corresponding to the cycles of g. This partition is returned in the form of a sequence Q of pairs, where the terms of Q correspond to the distinct cycle lengths of g. The value of the term $Q[i]$ is a tuple $< l_i, n_i >$ belonging to $\mathbf{Z} \times \mathbf{Z}$. Here l_i is the length of a cycle of g and n_i is the number of cycles of length l_i.

`Degree(g)`

Given a permutation g, return the degree of g, i.e. the number of points moved by g.

`IsEven(g)`

Returns `true` if the permutation g is an even permutation, `false` otherwise.

`Sign(g)`

Return 1 if the permutation g is even, return -1 if g is odd.

`Order(g)`

Order of the permutation g.

58.6.4 Predicates for Permutations

`g eq h`

Given permutations g and h belonging to the same generic group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given permutations g and h belonging to the same generic group, return `true` if g and h are distinct elements, `false` otherwise.

`IsId(g)`

`IsIdentity(g)`

Returns `true` if the permutation g is the identity permutation.

Example H58E9

We illustrate the permutation operations by applying them to some elements of Sym(9).

```
> G := Sym(9);
> x := G ! (1,2,4)(5,6,8)(3,9,7);
> y := G ! (4,5,6)(7,9,8);
> x*y;
(1, 2, 5, 4)(3, 8, 6, 7)
> x^-1;
(1, 4, 2)(3, 7, 9)(5, 8, 6)
> x^2;
(1, 4, 2)(3, 7, 9)(5, 8, 6)
> x / y;
(1, 2, 6, 9, 8, 4)(3, 7)
> x^y;
(1, 2, 5)(3, 8, 9)(4, 7, 6)
> (x, y);
(1, 7, 3, 6)(4, 5, 9, 8)
> x^y eq y^x;
false
> CycleStructure(x^2*y);
[ <6, 1>, <2, 1>, <1, 1> ]
> Degree(y);
6
> Order(x^2*y);
6
```

58.6.5 Set Operations

The creation of a base and strong generating set (BSGS) for a permutation group G provides us with a very compact representation of the set of elements of G. A particular BSGS imposes an order on the elements of G (lexicographic ordering of base images). It thus makes sense to talk about the 'number' of a group element relative to a particular BSGS.

`G * H`

Given permutation groups G and H, where G and H both belong to the same generic group, form the set product $\{g * h | g \in G, h \in H\}$ as a set of group elements.

`ElementSet(G, H)`

Given a group G and a subgroup H of G, return the elements of H in the form of a set of elements of G. This function is only applicable to very small groups.

`NumberingMap(G)`

A bijective mapping from the group G onto the set of integers $\{1 \ldots |G|\}$. The actual mapping depends upon the base and strong generating set chosen for G.

`RandomProcess(G)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 20

Create a process to generate randomly chosen elements from the finite group G. The process is based on the product-replacement algorithm of [CLGM+95], modified by the use of an accumulator. At all times, N elements are stored where N is the maximum of the specified value for `Slots` and `Ngens`$(G)+1$. Initially, these are just the generators of G. As well, one extra group element is stored, the accumulator. Initially, this is the identity. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call chooses one of the elements in the slots and multiplies it into the accumulator. The element in that slot is replaced by the product of it and another randomly chosen slot. The random value returned is the new accumulator value. Setting `Scramble := m` causes m such operations to be performed before the process is returned.

`Random(G: parameters)`

`Short`	BoolElt	*Default :* `false`

A randomly chosen element for the group G. If a BSGS is known for G, then the distribution will be uniform over G. If no BSGS is known, then the *random* element is chosen by multiplying out a *random* word in the generators. Since it is usually not practical to choose words long enough to properly sample the elements of G, the element returned will usually be biased. The boolean-valued parameter `Short` is used in this situation to indicate that a short word will suffice. Thus, if `Random` is invoked with `Short` assigned the value `true` then the element is constructed using a short word.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the finite group G, construct a random element of G by forming a random product over the expanded generating set constructed when the process was created. For large degree groups, or groups for which a BSGS is not known, this function should be used in preference to `Random(G)`.

`Representative(G)`

`Rep(G)`

An element chosen from the permutation group G.

Example H58E10

We use the function `NumberingMap` to construct the multiplication table for the dihedral group of order 12.

```
> G := DihedralGroup(GrpPerm, 6);
> f := NumberingMap(G);
> [ [ f(x*y) : y in G ] : x in G ];
[
    [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ],
    [ 2, 3, 4, 5, 6, 1, 12, 7, 8, 9, 10, 11 ],
    [ 3, 4, 5, 6, 1, 2, 11, 12, 7, 8, 9, 10 ],
    [ 4, 5, 6, 1, 2, 3, 10, 11, 12, 7, 8, 9 ],
    [ 5, 6, 1, 2, 3, 4, 9, 10, 11, 12, 7, 8 ],
    [ 6, 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 7 ],
    [ 7, 8, 9, 10, 11, 12, 1, 2, 3, 4, 5, 6 ],
    [ 8, 9, 10, 11, 12, 7, 6, 1, 2, 3, 4, 5 ],
    [ 9, 10, 11, 12, 7, 8, 5, 6, 1, 2, 3, 4 ],
    [ 10, 11, 12, 7, 8, 9, 4, 5, 6, 1, 2, 3 ],
    [ 11, 12, 7, 8, 9, 10, 3, 4, 5, 6, 1, 2 ],
    [ 12, 7, 8, 9, 10, 11, 2, 3, 4, 5, 6, 1 ]
]
```

Example H58E11

We illustrate the use of the function `Random` using the wreath product of the symmetric group of degree 4 and the cyclic group of order 6.

```
> G := WreathProduct( Sym(4), CyclicGroup(GrpPerm, 6));
> G;
Permutation group G acting on a set of cardinality 24
    (1, 5, 9, 13, 17, 21)(2, 6, 10, 14, 18, 22) (3, 7, 11, 15, 19, 23)
        (4, 8, 12, 16, 20, 24)
    (1, 2, 3, 4)
    (1, 2)
> Order(G);
```

```
1146617856
> Random(G);
(1, 17, 12, 4, 18, 10, 3, 20, 9, 2, 19, 11)(5, 22, 13, 6, 21, 15)
    (7, 24, 16)(8, 23, 14)
```

We display the cycle structures of 10 random elements of G.

```
> R := [ CycleStructure(Random(G)) : i in [1..10] ];
> R;
[
    [ <6, 1>, <3, 6> ],
    [ <9, 1>, <6, 2>, <3, 1> ],
    [ <9, 2>, <3, 2> ],
    [ <12, 1>, <9, 1>, <3, 1> ],
    [ <18, 1>, <6, 1> ],
    [ <18, 1>, <6, 1> ],
    [ <12, 1>, <6, 2> ],
    [ <6, 3>, <2, 3> ],
    [ <6, 1>, <4, 3>, <2, 3> ],
    [ <6, 3>, <3, 2> ]
]
```

58.7 Conjugacy

`Class(H, x)`

`Conjugates(H, x)`

Given a group H and an element x belonging to a group K such that H and K are subgroups of the same symmetric group, this function returns the set of conjugates of x under the action of H. If $H = K$, the function returns the conjugacy class of x in H.

`ConjugacyClasses(G: parameters)`

`Classes(G: parameters)`

Construct a set of representatives for the conjugacy classes of G. The classes are returned as a sequence of triples containing the element order, the class length and a representative element for the class. The parameters `Reps` and `Al` enable the user to select the algorithm that is to be used.

`Reps` [GrpPermElt] *Default :*

`Reps :=` Q: Create the classes of G by using the random algorithm but using the group elements in Q as the "random" elements tried. The element orders and lengths of the classes will be computed and checked for consistency.

`Reps` [<GrpPermElt, RngIntElt>]

`Reps :=` Q Create the classes of G assuming that the first elements of the tuples in Q form a complete set of conjugacy class representatives and the corresponding integer is the size of the conjugacy class. The only check performed is that the class sizes sum to the group order.

`Al`	MonStgElt	*Default :* "*Default*"
`WeakLimit`	RngIntElt	*Default :* 500
`StrongLimit`	RngIntElt	*Default :* 5000

`Al := "Action"`: Create the classes of G by computing the orbits of the set of elements of G under the action of conjugation. This option is only feasible for small groups.
`Al := "Random"`: Construct the conjugacy classes of elements for a permutation group G using an algorithm that searches for representatives of all conjugacy classes of G by examining a random selection of group elements and their powers. The behaviour of this algorithm is controlled by two associated optional parameters `WeakLimit` and `StrongLimit`, whose values are positive integers n_1 and n_2, say. Before describing the effect of these parameters, some definitions are needed: A mapping $f : G \to I$ is called a class invariant if $f(g) = f(g^h)$ for all $g, h \in G$. For permutation groups, the cycle structure of g is a readily computable class invariant. Two elements g and h are said to be *weakly conjugate* with respect to the class invariant f if $f(g) = f(h)$. By definition, conjugacy implies weak conjugacy, but the converse is false. The random algorithm first examines n_1 random elements and their powers, using a test for weak conjugacy. It then proceeds to examine a further n_2 random elements and their powers, using a test for ordinary conjugacy. The idea behind this strategy is that the algorithm should attempt to find as many classes as possible using the very cheap test for weak conjugacy, before employing the more expensive ordinary conjugacy test to recognize the remaining classes.
`Al := "Inductive"`: Use G. Butler's inductive method to compute the classes. See Butler [But94] for a description of the algorithm. The action and random algorithms are used by this algorithm to find the classes of any small groups it is called upon to deal with.
`Al := "Extend"`: Construct the conjugacy classes of G by first computing classes in a quotient G/N and then extending these classes to successively larger quotients G/H until the classes for $G/1$ are known. More precisely, a series of subgroups $1 = G_0 < G_1 < \cdots < G_r = R < G$ is computed such that R is the (solvable) radical of G and G_{i+1}/G_i is elementary abelian. The radical quotient G/R is computed and its classes and centralizers of their representatives found and pulled back to G. The parameters `TFAl` and `ASAl` control the algorithm used to compute the classes of G/R.

To extend from G/G_{i+1} to the next larger quotient G/G_i, an affine action of each centralizer on a quotient of the elementary abelian layer G_{i+1}/G_i is computed. Each distinct orbit in that action gives rise to a new class of the larger quotient (see Mecky and Neubuser [MN89]).

`Al := "Default"`: First some special cases are checked for: If `IsAltsym(G)` then the classes of G are computed from the partitions of `Degree(G)`. If G is solvable, an isomorphic representation of G as a pc-group is computed and the classes computed in that representation. In general, the action algorithm will be used if $|G| \leq 5000$, otherwise the extension algorithm will be applied.

`TFAl` MonStgElt *Default :* "*Default*"

This parameter controls the algorithm used to compute the classes of a group with trivial Fitting subgroup, such as the group G/R in the description of the `"Extend"` method. The possible settings are the same as for `Al`, except that `"Extend"` is not a valid choice. The `"Action"`, `"Random"` and `"Inductive` settings behave as described above. The default algorithm is Derek Holt's generalisation of the Cannon/Souvignier fusion method to all classes of the group. The original fusion algorithm used fusion only on classes within a direct product normal subgroup, see [CS97]. Holt has extended the use of fusion to all conjugacy classes, avoiding the random part of the Cannon/Souvignier method. This algorithm must use another algorithm to find the classes of almost simple groups arising from the socle of the TF-group. The algorithm used for this is controlled by the parameter `ASAl`.

`ASAl` MonStgElt *Default :* "*Default*"

This parameter controls the algorithm used to compute the classes of an almost simple group from within the default TF-group algorithm. The possibilities for this parameter are as for `TFAl`. The default algorithm first determines if `Altsym(G)` is true. If so, classes are deduced from the partitions of `Degree(G)`. Next, if the order of G is ≤ 5000 then the action algorithm is used. If the socle of G has the correct order to be $PSL(2, q)$, for some q, then the random algorithm is used on G. Otherwise the inductive algorithm is used.

`Centralisers` BoolElt *Default :* `false`

A flag to force the storing of the centralisers of the class representatives with the class table. This does not apply to the action algorithm. In the case of the extension algorithm, this will do extra work to lift the centralisers through the final elementary abelian layer.

`PowerMap` BoolElt *Default :* `false`

A flag to force the storing of whatever power map information is produced by the classes algorithm used. In the case of the extension algorithm, this flag forces the computation of the full power map en-route, and may take considerably longer than not doing so. However, it is overall faster to set this flag true when the `PowerMap` is going to be computed anyway.

`ClassRepresentative(G, x)`

Given a group G for which the conjugacy classes are known and an element x of G, return the designated representative for the conjugacy class of G containing x.

`ClassCentraliser(G, i)`

The centraliser of the representative element stored for conjugacy class number i in group G. The group computed is stored with the class table for reference by future calls to this function.

`ClassMap(G:` *parameters*`)`

Given a group G, construct the conjugacy classes and the class map f for G. For any element x of G, $f(x)$ will be the index of the conjugacy class of x in the sequence returned by the `Classes` function. If the parameter `Orbits` is set `true`, the classes are computed as orbits of elements under conjugation and the class map is stored as a list of images of the elements of G (a list of length $|G|$). This option gives fast evaluation of the class map but is practical only in the case of very small groups. With `Orbits := false`, `WeakLimit` and `StrongLimit` are used to control the random classes algorithm (see function `Classes`).

`IsConjugate(G, g, h:` *parameters*`)`

Given a group G and elements g and h belonging to G, return the value `true` if g and h are conjugate in G. The function returns a second value if the elements are conjugate: an element k which conjugates g into h. The method used is the backtrack search of Leon [Leo97]. This search may be speeded considerably by knowledge of (subgroups of) the centralizers of g and h in G. The parameters relate to these subgroups.

`Centralizer`	MonStgElt	*Default :* "*Default*"
`LeftSubgroup`	GrpPerm	*Default :* $\langle g \rangle$
`RightSubgroup`	GrpPerm	*Default :* $\langle h \rangle$

The `LeftSubgroup` and `RightSubgroup` parameters enable the user to supply known subgroups of the centralizers of g and h respectively to the algorithm. By default, the cyclic subgroups generated by g and h are the known subgroups. The `Centralizer` parameter controls whether the algorithm starts by computing one or both centralizers in full before the conjugacy test. The `"Default"` behaviour is to compute the left centralizer, i.e. $C_G(g)$, unless either a left or right subgroup is specified, in which case no centralizer calculation is done. Other possible values are the four possibilities `"Left"` which forces computation of $C_G(g)$, `"Right"` which forces $C_G(h)$ to be computed, `"Both"` which computes both centralizers, and `"None"` which will compute no centralizers.

`IsConjugate(G, H, K:` *parameters*`)`

Given a group G and subgroups H and K of G, return the value `true` if H and K are conjugate in G. The function returns a second value if the subgroups are conjugate: an element z which conjugates H into K. The method used is the backtrack search of Leon [Leo97].

`Compute`	MonStgElt	*Default :* "*Default*"

This parameter may be set to any of "Both", "Default", "Left", "None", or "Right". This controls which normalisers are computed before starting the conjugacy test. The default strategy is currently "Left", which computes the normalizer of H in G before starting the conjugacy test between H and K. The greater the difference between H and K and their normalizers in G, the more helpful to the search it is to compute their normalizers.

`LeftSubgroup`	GrpPerm	*Default :* H
`RightSubgroup`	GrpPerm	*Default :* K

Instead of having the `IsConjugate` function compute the normalizers of H and K, the user may supply any known subgroup of G normalizing H (as `LeftSubgroup`) or normalizing K (as `RightSubgroup`) to be used as the normalizing groups to aid the search.

`Exponent(G)`

The exponent of the group G. This is computed as the product of the exponents of the Sylow subgroups fo G.

`NumberOfClasses(G)`

`Nclasses(G)`

The number of conjugacy classes of elements for the group G.

`PowerMap(G)`

Given a group G, construct the power map for G. Suppose that the order of G is m and that G has r conjugacy classes. When the classes are determined by Magma, they are numbered from 1 to r. Let C be the set of class indices $\{1, \ldots, r\}$ and let Z be the set of integers. The *power map* f for G is the mapping

$$f : C \times Z \to C$$

where the value of $f(i, j)$ for $i \in C$ and $j \in Z$ is the number of the class which contains x_i^j, where x_i is a representative of the i-th conjugacy class.

`AssertAttribute(G, "Classes", Q)`

Assert the class representatives of G. The action taken is identical to using the `ConjugacyClasses` function described above, with the parameter `Reps` set to Q. Thus Q may be a sequence of group elements or a sequence of tuples giving class representatives and class lengths.

Example H58E12

The conjugacy classes of the Mathieu group M_{11} can be constructed as follows:

```
> SetSeed(2);
> M11 := sub<Sym(11) | (1,10)(2,8)(3,11)(5,7), (1,4,7,6)(2,11,10,9)>;
> Classes(M11);
Conjugacy Classes of group M11
------------------------------
[1]     Order 1       Length 1
        Rep Id(M11)

[2]     Order 2       Length 165
        Rep (3, 10)(4, 9)(5, 6)(8, 11)

[3]     Order 3       Length 440
        Rep (1, 2, 4)(3, 5, 10)(6, 8, 11)

[4]     Order 4       Length 990
        Rep (3, 6, 10, 5)(4, 8, 9, 11)

[5]     Order 5       Length 1584
        Rep (1, 3, 6, 2, 8)(4, 7, 10, 9, 11)

[6]     Order 6       Length 1320
        Rep (1, 11, 2, 6, 4, 8)(3, 10, 5)(7, 9)

[7]     Order 8       Length 990
        Rep (1, 4, 5, 6, 2, 7, 11, 10)(8, 9)

[8]     Order 8       Length 990
        Rep (1, 7, 5, 10, 2, 4, 11, 6)(8, 9)

[9]     Order 11      Length 720
        Rep (1, 11, 9, 10, 4, 3, 7, 2, 6, 5, 8)

[10]    Order 11      Length 720
        Rep (1, 9, 4, 7, 6, 8, 11, 10, 3, 2, 5)
```

Example H58E13

The default values for the random class algorithm are adequate for a large variety of groups. We look at what happens when we vary the parameters in the case of the Higman-Sims simple group represented on 100 letters. In this case the default strategy reduces to a random search. The first choice of parameters does not look at enough random elements. Increasing the limit on the number of random elements examined will ensure the algorithm succeeds.

```
> G := sub<Sym(100) |
```

```
>    (2,8,13,17,20,22,7)(3,9,14,18,21,6,12)(4,10,15,19,5,11,16)
>        (24,77,99,72,64,82,40)(25,92,49,88,28,65,90)(26,41,70,98,91,38,75)
>        (27,55,43,78,86,87,45)(29,69,59,79,76,35,67)(30,39,42,81,36,57,89)
>        (31,93,62,44,73,71,50)(32,53,85,60,51,96,83)(33,37,58,46,84,100,56)
>        (34,94,80,61,97,48,68)(47,95,66,74,52,54,63),
>    (1,35)(3,81)(4,92)(6,60)(7,59)(8,46)(9,70)(10,91)(11,18)(12,66)(13,55)
>        (14,85)(15,90)(17,53)(19,45)(20,68)(21,69)(23,84)(24,34)(25,31)(26,32)
>        (37,39)(38,42)(40,41)(43,44)(49,64)(50,63)(51,52)(54,95)(56,96)(57,100)
>        (58,97)(61,62)(65,82)(67,83)(71,98)(72,99)(74,77)(76,78)(87,89) >;
>  K := Classes(G:WeakLimit := 20, StrongLimit := 50);
Runtime error in 'Classes': Random classes algorithm failed
> K := Classes(G: WeakLimit := 20, StrongLimit := 100);
> NumberOfClasses(G);
24
```

As the group has only 24 classes, the first random search could have succeeded by looking at 50 elements. On this occasion it did not, but looking at 100 elements did succeed.
We print the order, length and cycle structure for each conjugacy class.

```
> [ < k[1], k[2], CycleStructure(k[3]) > : k in K ];
[
       <1, 1, [ <1, 100> ]>,
       <2, 5775, [ <2, 40>, <1, 20> ]>,
       <2, 15400, [ <2, 50> ]>,
       <3, 123200, [ <3, 30>, <1, 10> ]>,
       <4, 11550, [ <4, 20>, <2, 10> ]>,
       <4, 173250, [ <4, 20>, <2, 6>, <1, 8> ]>,
       <4, 693000, [ <4, 20>, <2, 8>, <1, 4> ]>,
       <5, 88704, [ <5, 20> ]>,
       <5, 147840, [ <5, 20> ]>,
       <5, 1774080, [ <5, 19>, <1, 5> ]>,
       <6, 1232000, [ <6, 15>, <2, 5> ]>,
       <6, 1848000, [ <6, 12>, <3, 6>, <2, 4>, <1, 2> ]>,
       <7, 6336000, [ <7, 14>, <1, 2> ]>,
       <8, 2772000, [ <8, 10>, <4, 4>, <2, 2> ]>,
       <8, 2772000, [ <8, 10>, <4, 3>, <2, 3>, <1, 2> ]>,
       <8, 2772000, [ <8, 10>, <4, 4>, <2, 2> ]>,
       <10, 2217600, [ <10, 8>, <5, 4> ]>,
       <10, 2217600, [ <10, 10> ]>,
       <11, 4032000, [ <11, 9>, <1, 1> ]>,
       <11, 4032000, [ <11, 9>, <1, 1> ]>,
       <12, 3696000, [ <12, 6>, <6, 3>, <4, 2>, <2, 1> ]>,
       <15, 2956800, [ <15, 6>, <5, 2> ]>,
       <20, 2217600, [ <20, 4>, <10, 2> ]>,
       <20, 2217600, [ <20, 4>, <10, 2> ]>
]
```

We construct the power map and tabulate the second, third and fifth powers of each class.

```
> p := PowerMap(G);
```

```
> [ < i, p(i, 2), p(i, 3), p(i, 5) > : i in [1 .. #K] ];
[
    <1, 1, 1, 1>,
    <2, 1, 2, 2>,
    <3, 1, 3, 3>,
    <4, 4, 1, 4>,
    <5, 2, 5, 5>,
    <6, 2, 6, 6>,
    <7, 2, 7, 7>,
    <8, 8, 8, 1>,
    <9, 9, 9, 1>,
    <10, 10, 10, 1>,
    <11, 4, 3, 11>,
    <12, 4, 2, 12>,
    <13, 13, 13, 13>,
    <14, 7, 14, 14>,
    <15, 6, 15, 15>,
    <16, 7, 16, 16>,
    <17, 8, 17, 2>,
    <18, 9, 18, 3>,
    <19, 20, 19, 19>,
    <20, 19, 20, 20>,
    <21, 12, 5, 21>,
    <22, 22, 9, 4>,
    <23, 17, 23, 5>,
    <24, 17, 24, 5>
]
```

58.8 Subgroups

58.8.1 Construction of a Subgroup

`sub< G | L >`

Given the permutation group G, construct the subgroup H of G, generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) A sequence of n integers defining a permutation of G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic permutation group. The subgroup H will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of H consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list.

`ncl< G | L >`

Given the permutation group G, construct the subgroup H of G that is the *normal closure* of the subgroup H generated by the elements specified by the list L (see [BC82]), where the possibilities for L are the same as for the `sub`-constructor.

Example H58E14

The group PGL$(2,7)$ in its natural action on projective points is generated by the set of permutations $\{(1,2,3,4,5,6,7),(2,4,3,7,5,6),(1,8)(2,7)(3,4)(5,6)\}$. Using the above syntax, the group may be defined in any of the following ways:
(a) By means of a list of generating permutations written as products of cycles:

```
> PGL27 := sub< Sym(8) | (1,2,3,4,5,6,7), (2,4,3,7,5,6), (1,8)(2,7)(3,4)(5,6)>;
> PGL27;
Permutation group PGL27 acting on a set of cardinality 8
    (1, 2, 3, 4, 5, 6, 7)
    (2, 4, 3, 7, 5, 6)
    (1, 8)(2, 7)(3, 4)(5, 6)
```

(b) By means of a list of integer sequences representing generators:

```
> PGL27 := sub< Sym(8) |
>               [2,3,4,5,6,7,1,8], [1,4,7,3,6,2,5,8], [8,7,4,3,6,5,2,1] >;
```

(c) In terms of preassigned elements of the symmetric group of degree 8:

```
> S8 := Sym(8);
> a := S8!(1,2,3,4,5,6,7);
> b := S8!(2,4,3,7,5,6);
> c := S8!(1,8)(2,7)(3,4)(5,6);
> PGL27 := sub<S8 | a, b, c>;
```

(d) By means of a set of generators:

```
> S8 := Sym(8);
> gens := { S8 | (1,2,3,4,5,6,7), (2,4,3,7,5,6), (1,8)(2,7)(3,4)(5,6) };
> PGL27 := sub<S8 | gens>;
```

(e) By means of a sequence of generators:

```
> S8 := Sym(8);
> gens := [ S8 | (1,2,3,4,5,6,7), (2,4,3,7,5,6), (1,8)(2,7)(3,4)(5,6) ];
> PGL27 := sub<S8 | gens>;
```

Example H58E15

A representation H of a 2-generator transitive group G in its action on unordered pairs is constructed as follows:

```
> G := AlternatingGroup(7);
> deg1 := Degree(G);
> pairs := [ { i, j } : j in [i+1..deg1], i in [1..deg1-1] ];
> deg2 := #pairs;
> h1 := [ Position(pairs, pairs[i] ^ G.1): i in [1..deg2] ];
> h2 := [ Position(pairs, pairs[i] ^ G.2): i in [1..deg2] ];
> H := sub<Sym(deg2) | h1, h2>;
> H;
Permutation group H acting on a set of cardinality 21
 (2,3,4,5,6)(7,8,9,10,11)(12,16,19,21,15)(13,17,20,14,18),
 (1,7,2)(3,8,12)(4,9,13)(5,10,14)(6,11,15)
```

Example H58E16

We illustrate the `ncl`-constructor by using it to construct the derived subgroup of the Hessian group H. We exploit the fact that the derived subgroup may be obtained as the normal closure of the subgroup generated by the commutators of the generators of H.

```
> H := PermutationGroup< 9 | (1,2,4)(5,6,8)(3,9,7), (4,5,6)(7,9,8) >;
> Order(H);
216
> D := ncl< H | (H.1, H.2) >;
> D;
Permutation group D acting on a set of cardinality 9
Order = 72 = 2^3 * 3^2
    (1, 7, 3, 6)(4, 5, 9, 8)
    (2, 9, 3, 5)(4, 6, 7, 8)
    (2, 6, 3, 8)(4, 5, 7, 9)
```

58.8.2 Membership and Equality

`g in G`

Given a permutation g and a permutation group G, return `true` if g is an element of G, `false` otherwise.

`g notin G`

Given a permutation g and a permutation group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G`

Given a permutation group G and a set S of permutations belonging to a group H, where G and H belong the same generic group, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G`

Given a permutation group G and a set S of permutations belonging to a group H, where G and H belong the same generic group, return `true` if S is not a subset of G, `false` otherwise.

`H subset G`

Given permutation groups G and H belonging to the same generic group, return `true` if H is a subgroup of G, `false` otherwise.

`H notsubset G`

Given permutation groups G and H belonging to the same generic group, return `true` if H is not a subgroup of G, `false` otherwise.

`H eq G`

Given permutation groups G and H belonging to the same generic group, return `true` if G and H are the same group, `false` otherwise.

`H ne G`

Given permutation groups G and H belonging to the same generic group, return `true` if G and H are distinct groups, `false` otherwise.

58.8.3 Elementary Properties of a Subgroup

`Index(G, H)`

The index of the subgroup H in the group G. The index is returned as an integer. If the orders of G and H are not known, they will be computed.

`FactoredIndex(G, H)`

The index of the subgroup H in the group G. The index is returned as a factored integer. The format is the same as for `FactoredOrder`. If the orders of G and H are not known, they will be computed.

`IsCentral(G, H)`

Returns `true` if the subgroup H of the group G lies in the centre of G, `false` otherwise.

`IsNormal(G, H)`

Returns `true` if the subgroup H of the group G is a normal subgroup of G, `false` otherwise.

`IsSelfNormalizing(G, H)`

`IsSelfNormalising(G, H)`

Returns `true` if the subgroup H of the group G is self-normalizing in G, `false` otherwise.

`IsSubnormal(G, H)`

Returns `true` if the subgroup H of the group G is subnormal in G, `false` otherwise.

58.8.4 Standard Subgroups

Unless the order is already known, each of the functions in this family will create a base and strong generating set for the group if one does not already exist.

`H ^ g`

`Conjugate(H, g)`

Construct the conjugate $g^{-1} * H * g$ of the permutation group H by the permutation g. The group H and the element g must belong to the same symmetric group.

`H meet K`

Given groups H and K which belong to the same symmetric group, construct the intersection of H and K. The intersection is found using the backtrack search of J. Leon [Leo97].

`IntersectionWithNormalSubgroup(G, N)`

`Check` BoolElt *Default :* `true`

Given groups G and N which belong to the same symmetric group and so that G normalises N, construct the intersection of G and N. The algorithm used is that of Cooperman, Finkelstein and Luks [CFL89], which uses a permutation representation of double the degree of G and N. Setting `Check` to `false` suppresses checking that G normalises N.

`CommutatorSubgroup(G, H, K)`

`CommutatorSubgroup(H, K)`

Given groups H and K, both subgroups of the group G, construct the commutator subgroup of H and K in the group G. If K is a subgroup of H, then the group G may be omitted. The algorithm used is described in [BC82].

`Centralizer(G, g: parameters)`

`Centraliser(G, g: parameters)`

Construct the centralizer of the permutation g in the group G; g and G must belong to a common symmetric group. A backtrack search through G as described in [Leo97] is employed.

`Subgroup`	GrpPerm	*Default :*

The parameter `Subgroup` may be used to supply a known subgroup of the centralizer. This may speed the search.

`Centralizer(G, H)`

`Centraliser(G, H)`

Construct the centralizer of the group H in the group G; G and H must belong to a common symmetric group. A backtrack search through G as described in [Leo97] is employed.

`CentralizerOfNormalSubgroup(G, H)`

Given G and H, belonging to a common symmetric group, with the restriction that H is a normal subgroup of G, construct the centralizer of H in G. A polynomial-time reduction algorithm described in Beals [Bea93] is used.

`SectionCentraliser(G, H, K)`

`SectionCentralizer(G, H, K)`

Return the full preimage in G of the centralizer in G/K of H/K. H and K must be normal subgroups of G with K contained in H. An algorithm of Luks [Luk93] is employed which involves computing the core of a subgroup in a group having twice the degree of G.

`Core(G, H)`

Given a subgroup H of the permutation group G, construct the maximal normal subgroup of G that is contained in the subgroup H. The algorithm employs repeated conjugation and intersection using the backtrack search of Leon [Leo97].

`H ^ G`

`NormalClosure(G, H)`

Given a subgroup H of the permutation group G, construct the normal closure of H in G.

`Normalizer(G, H: parameters)`

`Normaliser(G, H: parameters)`

`Subgroup`	GrpPerm	*Default :* H
`Bound`	RngIntElt	*Default :*

Given a subgroup H of the group G, construct the normalizer of H in G. A backtrack search as described in Leon [Leo97] is employed.

The parameter `Subgroup` may be used to pass the search a known subgroup of the normalizer. The default value of the starting subgroup is H. If `Bound` is set, the search will be terminated once the normalizing group found has order at least equal to `Bound`. If this does not happen, the search will complete as normal.

`SymmetricNormalizer(G)`

`Subgroup`	GrpPerm	*Default :* H
`Bound`	RngIntElt	*Default :*

`SymmetricNormaliser(G)`

Given a permutation group G acting on the set X, return the normalizer of G in the symmetric group on X. The parameters are as for `Normalizer` above.

`SylowSubgroup(G, p)`

`Sylow(G, p)`

Given a group G and a prime p, construct a Sylow p-subgroup of G. The algorithm used is that of Cannon, Cox and Holt [CCH97].

Example H58E17

We illustrate the use of these functions by applying them to a group of degree 30.

```
> M := PermutationGroup< 30 |
>         (1,2,3)(4,14,8)(5,15,9)(6,13,7)(25,27,26),
>         (4,20,13)(5,21,14)(6,19,15)(16,17,18)(27,28,29),
>         (1, 15)(2, 13)(3, 14)(4, 22)(5, 23)(6, 24)(7, 18)(8, 16)
>           (9, 17)(10, 21)(11, 19)(12, 20)(25, 29)(26, 27)(28, 30) >;
> FactoredOrder(M);
[ <2, 8>, <3, 10>, <5, 1> ]
> S2 := SylowSubgroup(M, 2);
> S2;
Permutation group S2 acting on a set of cardinality 30
Order = 256 = 2^8
    (1, 10)(2, 11)(3, 12)(4, 8)(5, 9)(6, 7)(13, 19)(14, 20)(15, 21)
        (16, 22)(17, 23)(18, 24)
    (1, 24)(2, 22)(3, 23)(4, 14)(5, 15)(6, 13)(7, 19)(8, 20)(9, 21)
        (10, 18)(11, 16)(12, 17)
    (4, 8)(5, 9)(6, 7)(13, 19)(14, 20)(15, 21)
    (4, 14)(5, 15)(6, 13)(7, 19)(8, 20)(9, 21)(25, 26)(29, 30)
```

```
    (1, 4)(2, 5)(3, 6)(7, 12)(8, 10)(9, 11)(13, 23)(14, 24)(15, 22)
        (16, 21)(17, 19)(18, 20)(25, 26)
    (27, 28)(29, 30)
    (27, 29)(28, 30)
    (25, 26)(29, 30)
```

We try to find a second Sylow subgroup $S2a$ that has trivial intersection with $S2$.

```
> b := exists(t){ x : x in M | Order(S2 meet S2^x) eq 1 };
> b;
true
> S2a := S2^t;
> N := Normalizer(M, S2);
> N;
Permutation group N acting on a set of cardinality 30
Order = 768 = 2^8 * 3
    (4, 8)(5, 9)(6, 7)(13, 19)(14, 20)(15, 21)
    (4, 14)(5, 15)(6, 13)(7, 19)(8, 20)(9, 21)
    (1, 10)(2, 11)(3, 12)(4, 8)(5, 9)(6, 7)(13, 19)(14, 20)(15, 21)
        (16, 22)(17, 23)(18, 24)
    (1, 24)(2, 22)(3, 23)(4, 14)(5, 15)(6, 13)(7, 19)(8, 20)(9, 21)
        (10, 18)(11, 16)(12, 17)
    (1, 22, 12)(2, 23, 10)(3, 24, 11)(4, 21, 13)(5, 19, 14)(6, 20, 15)
        (7, 8, 9)(16, 17, 18)
    (27, 29)(28, 30)
    (1, 14, 24, 4)(2, 15, 22, 5)(3, 13, 23, 6)(7, 12, 19, 17)(8, 10, 20, 18)
        (9, 11, 21, 16)(29, 30)
    (4, 14)(5, 15)(6, 13)(7, 19)(8, 20)(9, 21)(25, 26)(29, 30)
    (27, 28)(29, 30)
```

Thus the Sylow 2-subgroup is normalized by an element of order 3.

58.8.5 Maximal Subgroups

`IsMaximal(G, H: parameters)`

`Al` MonStgElt *Default :* "*Subgroups*"

Returns `true` if the subgroup H of the group G is a maximal subgroup of G. The algorithm used depends on the value of the parameter `Al`. The default value `Subgroups` computes the maximal subgroups of G if the index of H in G is over 1000 and the maximal subgroups are computable.The subgroup H is then tested for conjugacy with each class found. In the other cases, or when the `Al` parameter is set to `CosetImage`, the function is evaluated by first calling `IsProbablyMaximal` and if that returns `true` then constructing the permutation representation of G on the cosets of H and testing this representation for primitivity.

IsProbablyMaximal(G, H: *parameters*)

Tries RngIntElt *Default :* 20

Given a group G and a subgroup H of G, this function performs a probabilistic test for the maximality of H in G. The test involves adjoining random elements of G to H and determining if the result G. If not, then `false` is returned, otherwise `true` s returned. The number of random elements used is controlled by the parameter `Tries`, which is set to 20 by default.

MaximalSubgroups(G: *parameters*)

Construct the sequence of maximal subgroup classes of G. This is equivalent to the command `Subgroups(G: Al := "Maximal")`. The same parameters as for `Subgroups` are available to limit the search. The algorithm is described in [CH04].

Example H58E18

The `Subgroups` family of commands can deal with fairly large groups. We look at the maximal subgroups of the group of the $4 \times 4 \times 4$ Rubik's cube. This group has order about 1.7×10^{55}.

```
> load rubik444;
Loading "/home/magma/libs/pergps/rubik444"
The automorphism group of the 4 x 4 x 4 Rubik cube.
The group is represented as a permutation group of degree 72.
Its order is
2^50 * 3^29 * 5^9 * 7^7 * 11^4 * 13^2 * 17^2 * 19^2 * 23^2.
Group: G
> time max := MaximalSubgroups(G);
Time: 100.559
> #max;
46
> [Index(G, x`subgroup) : x in max];
[ 51090942171709440000, 51090942171709440000, 9161680528000,
9161680528000, 4509264634875, 4509264634875, 316234143225,
316234143225, 96197645544, 96197645544, 1577585295,
1577585295, 2496144, 2496144, 1961256, 1961256, 1307504,
1307504, 346104, 346104, 42504, 42504, 2187, 1352078,
1352078, 735471, 735471, 134596, 134596, 10626, 10626, 2024,
2024, 120, 276, 276, 56, 24, 24, 105, 28, 8, 35, 2, 2, 2 ]
```

58.8.6 Conjugacy Classes of Subgroups

`SubgroupClasses(G: parameters)`

`Subgroups(G: parameters)`

Representatives for the conjugacy classes of subgroups for the group G. The subgroups are returned as a sequence of records where the i-th record contains:

(a) A representative subgroup H for the i-th conjugacy class (field name `subgroup`).

(b) The order of the subgroup (field name `order`).

(c) The number of subgroups in the class (field name `length`).

(d) [Optional] A presentation for H (field name `presentation`).

`Al` MonStgElt *Default :* "*All*"

`Al := "All"`: Construct all subgroups of G.
`Al := "Maximal"`: Only construct maximal subgroups of G. This option reduces the number of intersections with any elementary abelian layer that need be considered and eliminates the need to recursively apply the algorithm.
`Al := "Normal"`: Only construct normal subgroups of G. This option does not use database lookup to find the normal subgroups of the radical quotient of G and also reduces the number of intersections with any layer that need be considered.

`LayerSizes` SeqEnum *Default :* See below

`LayerSizes := [ 2, 5, 3, 4, 7, 3, 11, 2, 17, 1]` is equivalent to the default. When constructing an Elementary Abelian series for the group, attempt to split 2-layers of size gt 2^5, 3-layers of size gt 3^4, etc. The implied exponent for 13 is 2 and for all primes greater than 17 the exponent is 1.

`Series` SeqEnum *Default :* See below

Use the given elementary abelian series rather than constructing the default series. The first subgroup in the series must be the solvable radical of G. The subgroups must form a descending chain of normal subgroups of G, such that each quotient is elementary abelian. The last subgroup in the series must be either elementary abelian or trivial.

`Presentation` BoolElt *Default :* `false`

`Presentation := true`: Construct a presentation for each subgroup.

`OrderEqual` RngIntElt *Default :*

`OrderEqual := n`: Only construct subgroups having order equal to n.

`OrderDividing` RngIntElt *Default :*

`OrderDividing := n`: Only construct subgroups having order dividing n.

`OrderMultipleOf` RngIntElt *Default :*

`OrderMultipleOf := n`: Only construct subgroups having order a multiple of n.

`IndexLimit` RngIntElt *Default :*

`IndexLimit := n`: Only construct subgroups having index in G less than or equal to n.

`IsElementaryAbelian` BoolElt *Default :* `false`

`IsElementaryAbelian := true`: Only construct elementary abelian subgroups of G.

`IsCyclic` BoolElt *Default :* `false`

`IsCyclic := true`: Only construct cyclic subgroups of G.

`IsAbelian` BoolElt *Default :* `false`

`IsAbelian := true`: Only construct abelian subgroups of G.

`IsNilpotent` BoolElt *Default :* `false`

`IsNilpotent := true`: Only construct nilpotent subgroups of G.

`IsSolvable` BoolElt *Default :* `false`

`IsSolvable := true`: Only construct solvable subgroups of G.

`IsNotSolvable` BoolElt *Default :* `false`

`IsNotSolvable := true`: Only construct insolvable subgroups of G.

`IsPerfect` BoolElt *Default :* `false`

`IsPerfect := true`: Only construct perfect subgroups of G.

`IsRegular` BoolElt *Default :* `false`

`IsRegular := true`: Only construct regular subgroups of G.

`IsTransitive` BoolElt *Default :* `false`

`IsTransitive := true`: Only construct transitive subgroups of G.

The Algorithm: (See Cannon, Cox and Holt [CCH01]) This command proceeds by first constructing an elementary abelian series for G together with G's radical quotient Q. We first attempt to locate the quotient in a database of groups with trivial Fitting subgroup. This database contains all such groups of order up to 216 000, and all such which are perfect of order up to 1 000 000. If Q is found then either all its subgroups, or its maximal subgroups are read from the database. (In some cases only the maximal subgroups are stored.) If Q is not found then we attempt to find the maximal subgroups of Q using a method of Derek Holt. For this to succeed all simple factors of the socle of Q must be found in a second database which currently contains all simple groups of order less than 1.6×10^7, as well as M_{24}, HS, J_3, McL, $Sz(32)$ and $L_6(2)$. There are also special routines to handle numerous other groups. These include: A_n for $n \leq 999$, $L_2(q)$, $L_3(q)$, $L_4(q)$, $L_5(q)$, $L_6(q)$ and $L_7(q)$ for all q, $U_3(q)$ for q prime and $q = 8, 9, 16, 25$, $U_4(q)$ for $q = 4, 5, 7$, $S_4(q)$ for all odd q and even $q \leq 16$, $L_d(2)$ for $d \leq 14$, and the following groups: $U_6(2)$, $S_8(2)$, $S_{10}(2)$, $O_8^{\pm}(2)$, $O_{10}^{\pm}(2)$, $S_6(3)$, $O_7(3)$, $O_8^{-}(3)$, $G_2(4)$, $G_2(5)$, ${}^3D_4(2)$, ${}^2F_4(2)'$, Co_2, Co_3, He, Fi_{22}.

If we have only maximal subgroups of Q, and more are required, we apply the algorithm recursively to the maximal subgroups to determine all subgroups of Q. This may take some time.

The subgroups are then extended to the whole group by stepwise extension through each layer of the elementary abelian series. For each layer this involves determining all possible intersections of a subgroup with this layer and all extensions with this intersection.

The limitations are that the simple factors of the socle of Q must be in the database, which is limited as above. Further, it may take some time to construct all subgroups from the maximal subgroups first found, and, if there is a large elementary abelian layer, there will be many possible intersections, which could also make the algorithm prohibitively slow.

There are numerous parameters for this function which allow the user to place restrictions on which subgroup classes are constructed. Using these restrictions may help overcome the problems noted above.

`SubgroupsLift(G, A, B, Q`: *parameters*`)`

This function isolates one step of the extension process used by the `Subgroups` family of functions. Q is a sequence of records such as returned by `Subgroups`(G). A and B are normal subgroups of G with A/B elementary abelian. The records in Q are interpreted as subgroups of G/A, which are lifted to all possible corresponding subgroups of G/B, subject to the parameters given.

`LowIndexSubgroups(G, n`: *parameters*`)`

`LowIndexSubgroups(G, t`: *parameters*`)`

Returns a sequence of subgroups of G, each with index at most n. The sequence will contain one representative from each conjugacy class of G-subgroups satisfying the index constraint. The algorithm used is described in Cannon, Holt, Slattery & Steel [CHSS03].

The previous version of the algorithm is available by setting the parameter `Algorithm` to the string `"Subgroups"`. In this case the group G is subject to the same restrictions as the group input to the `Subgroups` function above.

In the second form t should be a pair of integers $\langle a, b\rangle$, and subgroups with index in the interval $[a, b]$ will be returned.

Other parameters are `Presentation` which may be set `true` to return a second sequence of presentations of the groups found, and `Print` which may be set to a positive integer to turn on diagnostic printing of the progress of the algorithms.

Example H58E19

With the `Subgroups` family of commands we can get the entire collection of (classes of) subgroups of a group G. We look at the double cover of M_{12}.

```
> load m12cover;
Loading "/home/magma/libs/pergps/m12cover"
```

```
The two-fold cover of the Mathieu group M12 on 24 letters.
Order is 190080 = 2^7 * 3^3 * 5 * 11.
Group: G
> time s := SubgroupClasses(G);
Time: 4.469
> #s;
293
```

This may be too many. The parameters allow us to restrict attention to a subset of the subgroups. We specify that the function is to return only the elementary abelian 2-subgroups of G.

```
> se := SubgroupClasses(G : IsElementaryAbelian := true,
>               OrderMultipleOf := 2);
> #se;
14
> se : Minimal;
Conjugacy classes of subgroups
------------------------------

[ 1]    Order 2            Length 1
        GrpPerm: $, Degree 24, Order 2
[ 2]    Order 2            Length 495
        GrpPerm: $, Degree 24, Order 2
[ 3]    Order 2            Length 495
        GrpPerm: $, Degree 24, Order 2
[ 4]    Order 4            Length 495
        GrpPerm: $, Degree 24, Order 2^2
[ 5]    Order 4            Length 495
        GrpPerm: $, Degree 24, Order 2^2
[ 6]    Order 4            Length 1485
        GrpPerm: $, Degree 24, Order 2^2
[ 7]    Order 4            Length 1980
        GrpPerm: $, Degree 24, Order 2^2
[ 8]    Order 4            Length 5940
        GrpPerm: $, Degree 24, Order 2^2
[ 9]    Order 8            Length 495
        GrpPerm: $, Degree 24, Order 2^3
[10]    Order 8            Length 495
        GrpPerm: $, Degree 24, Order 2^3
[11]    Order 8            Length 1485
        GrpPerm: $, Degree 24, Order 2^3
[12]    Order 8            Length 1980
        GrpPerm: $, Degree 24, Order 2^3
[13]    Order 8            Length 1980
        GrpPerm: $, Degree 24, Order 2^3
[14]    Order 16           Length 495
        GrpPerm: $, Degree 24, Order 2^4
```

Example H58E20

Using the `SubgroupLattice` function we obtain a representative subgroup for each conjugacy class together with the inclusion relations between subgroups.
WARNING: *Computing the inclusions is very time consuming and should only be performed for small groups.*

```
> G := PSL(2,9);
> time L := SubgroupLattice(G);
Time: 0.200
> L;
Partially ordered set of subgroup classes
-----------------------------------------
[ 1]  Order 1    Length 1   Maximal Subgroups:
---
[ 2]  Order 2    Length 45  Maximal Subgroups: 1
[ 3]  Order 3    Length 20  Maximal Subgroups: 1
[ 4]  Order 3    Length 20  Maximal Subgroups: 1
[ 5]  Order 5    Length 36  Maximal Subgroups: 1
---
[ 6]  Order 4    Length 15  Maximal Subgroups: 2
[ 7]  Order 4    Length 15  Maximal Subgroups: 2
[ 8]  Order 4    Length 45  Maximal Subgroups: 2
[ 9]  Order 6    Length 60  Maximal Subgroups: 2 3
[10]  Order 6    Length 60  Maximal Subgroups: 2 4
[11]  Order 9    Length 10  Maximal Subgroups: 3 4
[12]  Order 10   Length 36  Maximal Subgroups: 2 5
---
[13]  Order 8    Length 45  Maximal Subgroups: 6 7 8
[14]  Order 12   Length 15  Maximal Subgroups: 4 6
[15]  Order 12   Length 15  Maximal Subgroups: 3 7
[16]  Order 18   Length 10  Maximal Subgroups: 9 10 11
---
[17]  Order 24   Length 15  Maximal Subgroups: 10 13 14
[18]  Order 24   Length 15  Maximal Subgroups: 9 13 15
[19]  Order 36   Length 10  Maximal Subgroups: 8 16
[20]  Order 60   Length 6   Maximal Subgroups: 9 12 15
[21]  Order 60   Length 6   Maximal Subgroups: 10 12 14
---
[22]  Order 360  Length 1   Maximal Subgroups: 17 18 19 20 21
> NumberOfInclusions(L!5, L!20);
6
> L[5];
Permutation group acting on a set of cardinality 10
Order = 5
  (1, 8, 9, 3, 4)(2, 7, 5, 10, 6)
```

The order and class length of each class of subgroups is listed, along with the information about where to find the maximal subgroups of a member of this class. Further information about

inclusions is available from the lattice. We see that 6 members of class 5 are contained in any fixed member of class 20.

58.8.7 Classes of Subgroups Satisfying a Condition

`NormalSubgroups(G: parameters)`

Construct the sequence of normal subgroup classes of G. This is equivalent to `Subgroups(G: Al := "Normal")`. The same parameters as for `Subgroups` are available to limit the search.

`ElementaryAbelianSubgroups(G: parameters)`

Construct the sequence of elementary abelian subgroups of G. This is equivalent to `Subgroups(G: IsElementaryAbelian := true)`. The same parameters as for `Subgroups` are available to limit the search.

`CyclicSubgroups(G: parameters)`

Construct the sequence of cyclic subgroups of G. This is equivalent to `Subgroups(G: IsCyclic := true)`. The same parameters as for `Subgroups` are available to limit the search.

`AbelianSubgroups(G: parameters)`

Construct the sequence of abelian subgroups of G. Equivalent to `Subgroups(G: IsAbelian := true)`. The same parameters as for `Subgroups` are available to limit the search.

`NilpotentSubgroups(G: parameters)`

Construct the sequence of nilpotent subgroups of G. This is equivalent to `Subgroups(G: IsNilpotent := true)`. The same parameters as for `Subgroups` are available to limit the search.

`SolvableSubgroups(G: parameters)`

Construct the sequence of solvable subgroups of G. This is equivalent to `Subgroups(G: IsSolvable := true)`. The same parameters as for `Subgroups` are available to limit the search.

`PerfectSubgroups(G: parameters)`

Construct the sequence of perfect subgroups of G. Equivalent to `Subgroups(G: IsNotSolvable := true)`. The same parameters as for `Subgroups` are available to limit the search.

`NonsolvableSubgroups(G: parameters)`

Construct the sequence of insolvable subgroups of G. This is equivalent to `Subgroups(G: IsNotSolvable := true)`. The same parameters as for `Subgroups` are available to limit the search.

`SimpleSubgroups(G: parameters)`

Construct the sequence of non-abelian simple subgroup classes of G. This is equivalent to `Subgroups(G: Al := "Simple")`. The same parameters as for `Subgroups` are available to limit the search.

58.9 Quotient Groups

58.9.1 Construction of Quotient Groups

`quo< G | L >`

Given the permutation group G, construct the quotient group $Q = G/N$, where N is the normal closure of the subgroup of G generated by the elements specified by L. The clause L is a list of one or more items of the following types:

(a) A sequence of n integers defining a permutation of G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic permutation group. The function returns

(a) the quotient group Q, and

(b) the natural homomorphism $f : G \to Q$.

Currently, the quotient group is constructed via the regular representation of the quotient, so the application of this operator is restricted to the case where the index of N in G is small. The representation of the quotient group that is returned is the result of a degree reduction applied to the regular representation, so need not be regular. The generators of the quotient are images of the generators of G.

The second return value is the epimorphism from G to the resulting quotient group.

`G / N`

Given a normal subgroup N of the permutation group G, construct the quotient of G by N. Currently, the quotient group is constructed via the regular representation of the quotient, so the application of this operator is restricted to the case where the index of N in G is small. The representation of the quotient group that is returned is the result of a degree reduction applied to the regular representation, so need not be regular. The generators of the quotient are images of the generators of G.

Example H58E21

The quotient of Sym(4) by the Klein 4-group is constructed by the following statement:

```
> Q, f := quo< Sym(4) | (1,2)(3,4), (1,3)(2,4) >;
> Q;
Permutation group Q acting on a set of cardinality 3
Order = 6 = 2 * 3
    (2, 3)
    (1, 2)
```

58.9.2 Abelian, Nilpotent and Soluble Quotients

A number of standard quotients may be constructed. The method first constructs a presentation for the permutation group and then applies the appropriate fp-group algorithm.

`AbelianQuotient(G)`

The maximal abelian quotient G/G' of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism $\pi : G \to G/G'$ is returned as second value.

`ElementaryAbelianQuotient(G, p)`

The maximal p-elementary abelian quotient Q of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism $\pi : G \to Q$ is returned as second value.

`pQuotient(G, p, c)`

Given a permutation group G, a prime p and a positive integer c, construct a pc-presentation for the largest p-quotient P of G having lower exponent-p class at most c. If c is given as 0, then the limit 127 is placed on the class.

The function also returns the natural homomorphism π from G to P, a sequence S describing the definitions of the pc-generators of P and a flag indicating whether P is the maximal p-quotient of G.

The k-th element of S is a sequence of two integers, describing the definition of the k-th pc-generator $P.k$ of P as follows.

- If $S[k] = [0, r]$, then $P.k$ is defined via the image of $G.r$ under π.
- If $S[k] = [r, 0]$, then $P.k$ is defined via the power relation for $P.r$.
- If $S[k] = [r, s]$, then $P.k$ is defined via the conjugate relation involving $P.r^{P.s}$.

`NilpotentQuotient(G, c)`

This function returns the class c nilpotent quotient of G, together with the epimorphism π from G onto this quotient.

`SolvableQuotient(G)`

`SolubleQuotient(G)`

The function returns the largest soluble quotient S of the permutation group G together with the epimorphism $\pi : G \to S$.

Example H58E22

The soluble quotient of the wreath product of Sym(6) with the dihedral group of order 12 is easily constructed:

```
> G := WreathProduct( Sym(6), DihedralGroup(6));
> #G;
1671768834048000000
> SQ, phi := SolubleQuotient(G);
SQ;
GrpPC : SQ of order 768 = 2^8 * 3
PC-Relations:
    SQ.1^2 = SQ.5,
    SQ.2^2 = Id(SQ),
    SQ.3^2 = Id(SQ),
    SQ.4^2 = Id(SQ),
    SQ.5^3 = Id(SQ),
    SQ.6^2 = Id(SQ),
    SQ.7^2 = Id(SQ),
    SQ.8^2 = Id(SQ),
    SQ.9^2 = Id(SQ),
    SQ.2^SQ.1 = SQ.2 * SQ.5,
    SQ.3^SQ.1 = SQ.3 * SQ.4 * SQ.6 * SQ.8,
    SQ.4^SQ.1 = SQ.4 * SQ.9,
    SQ.4^SQ.2 = SQ.4 * SQ.6 * SQ.7 * SQ.8,
    SQ.5^SQ.2 = SQ.5^2,
    SQ.5^SQ.3 = SQ.5 * SQ.7,
    SQ.5^SQ.4 = SQ.5 * SQ.6 * SQ.8,
    SQ.6^SQ.1 = SQ.6 * SQ.8,
    SQ.6^SQ.2 = SQ.7 * SQ.8,
    SQ.6^SQ.5 = SQ.6 * SQ.7 * SQ.8 * SQ.9,
    SQ.7^SQ.1 = SQ.8,
    SQ.7^SQ.2 = SQ.9,
    SQ.7^SQ.5 = SQ.7 * SQ.9,
    SQ.8^SQ.1 = SQ.7 * SQ.9,
    SQ.8^SQ.2 = SQ.6 * SQ.9,
    SQ.8^SQ.5 = SQ.6 * SQ.9,
    SQ.9^SQ.1 = SQ.6 * SQ.8 * SQ.9,
    SQ.9^SQ.2 = SQ.7,
    SQ.9^SQ.5 = SQ.7
```

58.10 Permutation Group Actions

58.10.1 G-Sets

Let G be a group. A *G-set* is a pair (Y, f), where Y is a set and $f : Y \times G \to Y$ is a mapping such that

$$(a) \qquad f(f(y, g), h) = f(y, gh), \quad \text{for all } g, h \in G$$

and

$$(b) \qquad f(y, 1) = y, \quad \text{for all } y \in Y.$$

The mapping f defines the *action* of G on the set Y.

If G is defined as a permutation group acting on the set X and Y is another G-set then there is a homomorphism of G^X into G^Y.

We distinguish three types of G-set for a permutation group G. The set on which G is defined will be referred to as the *natural G-set* and the action of G on X as the *natural action* of G.

Let A be a set. A *derived set* of A is defined (recursively) as follows:

(i) A subset of A is a derived set;

(ii) A set of k-subsets of A is a derived set;

(iii) A set of k-sequences of A is a derived set;

(iv) A set of ordered partitions of A is a derived set;

(v) A subset of a cartesian product of derived sets of A is a derived set.

The natural action of G on X induces a natural action on the G-closure Y of any derived set of X. Such a set Y is also a G-set. For example, a subset of X is a G-set for G if and only if it is a union of orbits for G.

Finally, a *general G-set* is an arbitrary set Y with an action f satisfying the conditions (a) and (b).

The notion of a G-set enables the user to work with several different actions of G. Rather than having to always work with the image of G with respect to an action on a set Y, the user may specify the required operation in terms of G.

58.10.2 Creating a G-Set

`GSetFromIndexed(G, Y)`

Given a group G and an indexed set Y with the same cardinality as the natural G-set, return a G-set corresponding to the natural bijection between the labelling L (= Labelling(G)) of G and Y. Explicitly, the bijection is

$$\phi : L \to Y : l \mapsto Y[\text{Position}(L, l)].$$

Then the returned G-set is the set Y endowed with the action

$$f : Y \times G \to Y : (y, p) \mapsto \phi(p(\phi^{-1}(y))).$$

`GSet(G, X, Y)`

`GSet(G, Y)`

Return the smallest derived G-set containing Y as a subset under the action of G on X. If X is omitted, then the natural action will be assumed. In practice, the set Y is expanded until for each element y of the expanded Y, the image of y under each generator of G under the action described by X is also in Y. The action of G on Y is then the action induced by the action of G on X.

`GSet(G)`

Given a permutation group G, return the G-set corresponding to the natural action of G.

`GSet(G, Y, f)`

Construct the smallest G-set containing Y as a subset with the given action f. The map f must satisfy the requirements of a G-set action. In particular, the domain of f must be a superset of $Y \times G$, the codomain a superset of Y and the two conditions listed at the beginning of this section must be met.

`Action(Y)`

The map giving the action of the group on the G-set Y.

`Group(Y)`

The group associated with the G-set Y.

`Labelling(G)`

Given a permutation group G of degree n, return an indexed set giving the internal mapping of the natural G-set of G onto the set $\{1, \ldots, n\}$, where n is the degree of G.

`Degree(g, Y)`

`Degree(g)`

Given an element g of a permutation group G and a G-set Y, return the cardinality of the subset of Y consisting of points that are moved by g. If Y is omitted, the natural G-set X is assumed.

`Degree(G, Y)`

`Degree(G)`

Given a G-set Y, return the cardinality of Y. If Y is omitted, the natural G-set X is assumed.

`Support(g, Y)`

`Support(g)`

Given an element g of a permutation group G and a G-set Y, return the subset of Y consisting of points that are moved by g. If Y is omitted, the natural G-set X is assumed.

`Support(G, Y)`

`Support(G)`

Given a permutation group G and a G-set Y, return the subset of Y consisting of points that are moved by at least one element of G. If Y is omitted the natural G-set for G is assumed.

Example H58E23

We construct a G-set with a user defined action. Our example will take a group G and a normal subgroup N of index 4. The G-set will be the irreducible characters of N, with the usual G action obtained from permuting the elements of N by conjugation. As this is not a derived set we will define the action via a map.

```
> G := PGammaL(2, 9);
> N := PSL(2, 9);
> CT := CharacterTable(N);
> X := SequenceToSet(CT);
> XxG := CartesianProduct(X, G);
> f := map< XxG -> X | x :-> x[1]^x[2] >;
> Y := GSet(G, X, f);
```

This defines our G-set `Y`. The inertia group of a character is its stabilizer in this action. Let us compute an inertia group.

```
> chi := CT[2];
> I := Stabilizer(G, Y, chi);
> Index(G, I);
2
> [#o : o in Orbits(G, Y)];
[ 1, 1, 1, 2, 2 ]
```

We find that two of the characters have inertia groups of index 2 in G, while three are G-invariant.

58.10.3 Images, Orbits and Stabilizers

`x ^ g`

Given a permutation group G with natural G-set X and an object x which is an element of some derived G-set of X, find the image of x under G.

`Image(g, Y, y)`

`Image(g, y)`

Given a permutation g belonging to a group G, a G-set Y, and an element y of Y, find the image of y under g. If y is an element of some derived G-set of G, the set Y may be omitted.

`Fix(g, Y)`

`Fix(g)`

Given a permutation g belonging to a group G and a G-set Y, construct the fixed-point set of g in its action on Y. In the case in which Y is the natural G-set, Y may be omitted. The fixed-point set is returned as a subset of points of Y.

`Fix(G, Y)`

`Fix(G)`

The fixed-point set of the permutation group G in its action on the G-set Y (or the natural G-set for G if Y is omitted).

`x ^ G`

Given a permutation group G with natural G-set X and an element x belonging to some derived G-set of X, construct the orbit of x under G. The orbit is returned as a G-set.

`Cycle(e, x)`

Let e be an element of a permutation group defined as acting on a set containing x. Returns the set of images of x under repeated application of e as an indexed set with x the first element. This gives the cycle containing x in the disjoint cycle representation of e.

`CycleDecomposition(e)`

Let e be an element of a permutation group defined as acting on a set X. Returns a sequence of indexed sets partitioning X, each of which is a cycle of e. This gives the full disjoint cycle representation of e.

`Orbit(G, Y, y)`

`Orbit(G, y)`

Given a permutation group G, a G-set Y, and an element y belonging to Y, construct the orbit of y under G. The orbit is returned as a G-set. If y is an element of some derived G-set of G, the set Y may be omitted.

`Orbits(G, Y)`

`Orbits(G)`

Given a permutation group G and a G-set Y, construct the orbits of G on Y. If the set Y is omitted, the orbits of G on its natural G-set are constructed. The orbits are returned as a sequence of G-sets.

`OrbitRepresentatives(G)`

Given a permutation group G, construct the orbits of G on its natural G-set. The orbit descriptions are returned as a sequence of tuples $\langle l, r \rangle$ giving the length l and a representative r of each orbit of G on its support.

This function stores *only* the orbit representatives and so is more space-efficient than `Orbits`. However, it should be used only if the user wants to determine just the orbit representatives; queries about the orbits containing other elements of the support will cause further computation.

`OrbitClosure(G, Y, S)`

`OrbitClosure(G, S)`

Given a subset S of the G-set Y, construct the smallest G-invariant subset of Y that contains S. If Y is the natural G-set for G it may be omitted.

`IsConjugate(G, Y, y, z)`

`IsConjugate(G, y, z)`

Given elements y and z belonging either to a G-set Y or to a (restricted) derived set of Y, return the value `true` if there exists an element $g \in G$ such that $y^g = z$. Otherwise, return `false`. If such an element exists, then it is returned as the second value of the function. If y and z belong to the natural G-set, then Y may be omitted. Currently, y and z are restricted to being elements, sets of elements, multisets of elements, sequences of elements, ordered partitions, or unordered partitions of Y.

`Stabilizer(G, Y, y)`

`Stabiliser(G, Y, y)`

`Stabilizer(G, y)`

`Stabiliser(G, y)`

Given a permutation group G and a G-set Y, and an object y which is either an element, a sequence of elements, a set of elements, an ordered partition or a tuple over the G-set Y, find the stabilizer of y in G. The stabilizer is returned as a subgroup of G. If Y is the natural G-set, it may be omitted.

`IsPrimitive(G, Y)`

`IsPrimitive(G)`

Returns `true` if G acts primitively on the G-set Y. If Y is the natural G-set, the set Y may be omitted.

`IsTransitive(G, Y)`

`IsTransitive(G)`

Returns `true` if G acts transitively on the G-set Y. If Y is the natural G-set, the set Y may be omitted.

`IsTransitive(G, Y, k)`

`IsTransitive(G, k)`

Returns `true` if G acts k-transitively on the G-set Y. If Y is the natural G-set, the set Y may be omitted.

`IsSharplyTransitive(G, Y, k)`

`IsSharplyTransitive(G, k)`

Returns `true` if G acts sharply k-transitively on the G-set Y. If Y is the natural G-set, the set Y may be omitted.

`Transitivity(G, Y)`

`Transitivity(G)`

The degree of transitivity of G acting on the G-set Y. The set Y may be omitted if it is the same as the natural G-set.

`IsRegular(G, Y)`

`IsRegular(G)`

Returns `true` if G acts regularly on the G-set Y. If Y is the natural G-set, the set Y may be omitted. The algorithm used is that of Sims, see [CB92].

`IsSemiregular(G, Y)`

`IsSemiregular(G)`

Returns `true` if G acts semiregularly on the G-set Y. If Y is the natural G-set, the set Y may be omitted. The algorithm used is a variation of Sims' regularity test, see [CB92].

`IsSemiregular(G, Y, S)`

`IsSemiregular(G, S)`

Given a permutation group G, a G-set Y for G, and a union of orbits S for G in its action on Y, return `true` if G acts semiregularly on S. If Y is the natural G-set, then Y may be omitted.

`IsFrobenius(G)`

Returns `true` if the permutation group G is a Frobenius group with respect to its natural action, `false` otherwise. (A group G defined as acting on X is Frobenius if it acts transitively but non-regularly on X and if the pointwise stabilizer of any two distinct points of X is the trivial group.)

Example H58E24

We apply some of these functions to the Mathieu group M_{24}, taking as generators the following three permutations:

$$(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,24),$$

$$(2,16,9,6,8)(3,12,13,18,4)(7,17,10,11,22)(14,19,21,20,15),$$

$$(1,22)(2,11)(3,15)(4,17)(5,9)(6,19)(7,13)(8,20)(10,16)(12,21)(14,18)(23,24).$$

```
> M24 := sub< Sym(24) |
>  (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,24),
>  (2,16,9,6,8)(3,12,13,18,4)(7,17,10,11,22)(14,19,21,20,15),
>  (1,22)(2,11)(3,15)(4,17)(5,9)(6,19)(7,13)(8,20)(10,16)(12,21)(14,18)(23,24)>;
> M24;
Permutation group M24 acting on a set of cardinality 24
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
       21, 22, 24)
    (2, 16, 9, 6, 8)(3, 12, 13, 18, 4)(7, 17, 10, 11, 22)(14, 19, 21, 20, 15)
    (1, 22)(2, 11)(3, 15)(4, 17)(5, 9)(6, 19)(7, 13)(8, 20)(10, 16)(12, 21)
       (14, 18)(23, 24)
```

Choosing a random element x of M_{24}, we use it to compute some images.

```
> x := Random(M24);
> 1^x;
7
> [1,2,3,4]^x;
```

```
[ 7, 9, 8, 17 ]
> { 1,2,3,4}^x;
{ 17, 7, 8, 9 }
```

We find the stabilizer of the point 1, which is the group M_{23}.

```
> S1 := Stabilizer(M24, 1);
> S1;
Permutation group S1 acting on a set of cardinality 24
Order = 10200960 = 2^7 * 3^2 * 5 * 7 * 11 * 23
    (2, 16, 9, 6, 8)(3, 12, 13, 18, 4)(7, 17, 10, 11, 22)(14, 19, 21, 20, 15)
    (7, 17, 22)(8, 11, 13)(9, 14, 12)(10, 20, 19)(15, 23, 18)(16, 21, 24)
    (3, 6, 18)(5, 16, 14)(7, 21, 22)(8, 19, 17)(9, 20, 24)(11, 12, 13)
    (6, 18, 15)(7, 19, 16)(8, 13, 11)(9, 10, 22)(12, 21, 20)(14, 17, 24)
    (4, 12, 6, 19)(5, 22, 24, 8)(7, 17, 20, 14)(9, 15, 13, 18)(10, 21)(11, 16)
    (6, 22, 7)(8, 13, 11)(9, 20, 16)(10, 18, 21)(12, 15, 19)(14, 24, 23)
    (5, 12, 21)(6, 15, 18)(7, 22, 8)(9, 16, 17)(10, 14, 13)(11, 24, 19)
```

We next compute the stabilizer of the sequence $[1, 2, 3, 4, 5]$.

```
> SQ := Stabilizer(M24, [1,2,3,4,5]);
> SQ;
Permutation group SQ acting on a set of cardinality 24
Order = 48 = 2^4 * 3
    (6, 18, 15)(7, 19, 16)(8, 13, 11)(9, 10, 22)(12, 21, 20)(14, 17, 24)
    (7, 17, 22)(8, 11, 13)(9, 14, 12)(10, 20, 19)(15, 23, 18)(16, 21, 24)
    (6, 22, 7)(8, 13, 11)(9, 20, 16)(10, 18, 21)(12, 15, 19)(14, 24, 23)
> Orbits(SQ);
[
    GSet{@ 1 @},
    GSet{@ 2 @},
    GSet{@ 3 @},
    GSet{@ 4 @},
    GSet{@ 5 @},
    GSet{@ 6,18, 22, 15, 21, 9, 7, 23, 19, 20, 24, 10,
    14, 17, 16 12 @},
    GSet{@ 8, 13, 11 @}
]
```

The five fixed points together with the orbit of length 3 form a block of a $5-(24, 8, 1)$ design. By computing the orbit of this block under M_{24}, we obtain all the blocks of the design.

```
> B := { 1,2,3,4,5,8,11,13};
> D := B^M24;
> #D;
759
```

Finally, we check that the set stabilizer of the block $\{ 1, 2, 3, 4, 5, 8, 11, 13\}$ has index 759 in M_{24}.

```
> Index(M24, Stabilizer(M24, { 1,2,3,4,5,8,11,13}));
759
```

58.10.4 Action on a G-Space

`Action(G, Y)`

Given a permutation group G defined to be acting on X and a set Y, construct the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the set Y. The function returns:

(a) The natural homomorphism $\phi : G \to L$;

(b) The induced group L;

(c) The kernel of the action (a subgroup of G).

`ActionImage(G, Y)`

Given a permutation group G defined to be acting on X and a set Y, construct the permutation group L giving the action of G on the set Y.

`ActionKernel(G, Y)`

Construct the kernel of the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the G-set Y.

`IsFaithful(G, Y)`

Returns `true` if the action of G on the G-set Y is faithful.

Example H58E25

We take the group PSL$(3, 4)$ acting on projective points and construct its representation on flags (point-line pairs). In order to construct the flags, we need to find a line. If H is the stabilizer of a point α in PSL$(3, 4)$ in its action on projective points, then a line consists of α together with the points in any non-trivial orbit of $O_2(G)$.

```
> G := ProjectiveSpecialLinearGroup(3, 4);
> O2 := pCore( Stabilizer(G, 1), 2 );
> O2;
Permutation group O2 acting on a set of cardinality 21
Order = 16 = 2^4
       (3, 4)(5, 7)(9, 16)(10, 17)(11, 15)(13, 18)(14, 19)(20, 21)
       (3, 20)(4, 21)(5, 15)(7, 11)(9, 10)(13, 19)(14, 18)(16, 17)
       (2, 8)(5, 15)(6, 12)(7, 11)(9, 17)(10, 16)(13, 18)(14, 19)
       (2, 12)(5, 11)(6, 8)(7, 15)(9, 16)(10, 17)(13, 19)(14, 18)
> flag := < 1, Orbit(O2, 2) >;
> flag;
<1, GSet{@ 2, 6, 8, 12 @}>
> flags := GSet(G, Orbit(G, flag));
> #flags;
105
> GOnFlags := ActionImage(G, flags);
> GOnFlags;
Permutation group GOnFlags acting on a set of
```

```
cardinality 105
Order = 20160 = 2^6 * 3^2 * 5 * 7
> Stabilizer(GOnFlags, Rep(flags));
Permutation group acting on a set of cardinality 105
Order = 192 = 2^6 * 3
```

58.10.5 Action on Orbits

The operations described here are concerned with the class of G-sets consisting of G-invariant subsets of the natural G-set. Because of the special nature of such G-sets, more efficient algorithms are available for computing with homomorphisms of G induced by the action of G on such a G-set. See Butler [But85] for more details.

`OrbitAction(G, T)`

The homomorphism $f : G \to L$ induced by the action of G on the G-invariant subset T of X (a union of orbits).

`OrbitImage(G, T)`

The group L defined by the action of G on the G-invariant subset T of X (a union of orbits).

`OrbitKernel(G, T)`

The kernel of the homomorphism $f : G \to L$, where the group L gives the action of G on the G-invariant subset T of X (a union of orbits).

`IsOrbit(G, S)`

Returns `true` if the subset S of Support(G) is invariant under G.

Example H58E26

We study an intransitive group of degree 36 generated by the permutations

$$(3, 17, 26)(4, 16, 25)(5, 18, 27)(8, 15, 24),$$
$$(1, 32, 10)(2, 31, 11)(3, 35, 12)(6, 30, 15),$$
$$(12, 33, 24)(13, 29, 20)(14, 28, 19)(17, 30, 21),$$
$$(6, 26, 33)(7, 22, 34)(8, 21, 35)(9, 23, 36).$$

```
> G := PermutationGroup< 36 | (3, 17, 26)(4, 16, 25)(5, 18, 27)(8, 15, 24),
>                             (1, 32, 10)(2, 31, 11)(3, 35, 12)(6, 30, 15),
>                             (12, 33, 24)(13, 29, 20)(14, 28, 19)(17, 30, 21),
>                             (6, 26, 33)(7, 22, 34)(8, 21, 35)(9, 23, 36) >;
> IsTransitive(G);
false
> Orbit(G, 1);
```

```
GSet{@ 1, 32, 10 @}
> O := Orbits(G);
> O;
[
    GSet{@ 1, 32, 10 @},
    GSet{@ 2, 31, 11 @},
    GSet{@ 4, 16, 25 @},
    GSet{@ 5, 18, 27 @},
    GSet{@ 7, 22, 34 @},
    GSet{@ 9, 23, 36 @},
    GSet{@ 13, 29, 20 @},
    GSet{@ 14, 28, 19 @}
    GSet{@ 3, 17, 35, 26, 30, 12, 8, 33, 15, 21, 24, 6 @},
]
> Order(G);
933120
```

We see that the group is intransitive having eight orbits of size 3 and one orbit of size 12. We consider the action of G on the orbit of size 12.

```
> f := OrbitAction(G, O[9]);
> f;
Mapping from: GrpPerm: G to GrpPerm: $
> Im := Image(f);
> Im;
Permutation group acting on a set of cardinality 12
Order = 11520 = 2^8 * 3^2 * 5
    (1, 6, 9)(3, 5, 8)
    (4, 11, 8)(6, 10, 7)
    (6, 8)(7, 11)
    (2, 8, 10)(4, 12, 6)
    (3, 10, 8)(4, 6, 9)
> Ker := Kernel(f);
> Ker;
Permutation group acting on a set of cardinality 36
Order = 81 = 3^4
    (4, 16, 25)(5, 18, 27)
    (7, 22, 34)(9, 23, 36)
    (13, 29, 20)(14, 28, 19)
    (1, 32, 10)(2, 31, 11)(4, 25, 16)(5, 27, 18)
> IsElementaryAbelian(Ker);
true
```

Thus G has an elementary abelian normal subgroup of order 81 which is the kernel of the restriction of G to the orbit of size 12.

58.10.6 Action on a G-invariant Partition

This section describes the functions supplied by MAGMA for computing with block systems for a permutation group.

`IsBlock(G, S)`

Given a transitive permutation group G with natural G-set X, and a subset S of X, return `true` if S is a block for G in its action on X.

`IsPrimitive(G)`

Returns `true` if the transitive permutation group G is primitive.

`MaximalPartition(G)`

Construct a G-invariant partition P for the transitive permutation group G with natural G-set X. The partition P is maximal in the sense that there is no G-invariant partition P' of X such that some block of P' properly contains a block of the partition P. The block system is returned as a partition of X. If G is primitive, the partition with one block is returned.

`MinimalPartition(G: parameters)`

Construct a non-trivial G-invariant partition P of the natural G-set X of the transitive permutation group G. The partition P is minimal in the sense that there is no G-invariant partition P' of X such that some block of P' is properly contained in some block of the partition P. The block system is returned as a partition of X. If G is primitive, or if no partition satisfying the side-conditions (see below) is found, then the empty set is returned. The algorithm used is based on Schönert & Seress [SS94].

`Block := S` { ELT } *Default :* []

If S is non-empty, then the partition P must possess a block B such that S is a subset of B. In this case the algorithm used is that of Atkinson, [Atk75].

`MinimalPartitions(G: parameters)`

Construct all non-trivial minimal G-invariant partitions of the natural G-set X of the transitive permutation group G. A partition P is minimal in the sense that there is no G-invariant partition P' of X such that some block of P' is properly contained in some block of the partition P.

The minimal block systems are returned as a sequence of sets of sets. If G is primitive, the function returns the empty sequence.

The algorithm used is based on Schönert & Seress [SS94].

`Limit := n` RNGINTELT *Default :* ∞

The function will return after creating at most n block systems. This option is useful in situations where, say, two distinct minimal blocks systems are required for a reduction algorithm.

`AllPartitions(G)`

Construct all non-trivial G-invariant partitions of the natural G-set X of the transitive permutation group G. The structure returned is a set containing one block from each such partition. The block chosen is the block containing the first element of `Labelling(G)`.

`BlocksAction(G, P)`
`BlocksAction(G, P)`
`BlocksAction(G, P)`
`BlocksAction(G, P)`

Given a transitive permutation group G with natural G-set X and a G-invariant partition P of X, construct the group L induced by the action of G on the blocks of P. In the second form, P is specified by giving a single block of the partition. The function returns

(a) The natural homomorphism $f : G \to L$;

(b) The induced group;

(c) The kernel of the action (a subgroup of G).

The relationship between the supports of G and L is given by the returned mapping, which may also be used as a map from `Labelling(G)` to `Labelling(L)`. In the forward direction this takes each element in the support of G to its block number in the support of L, while in the reverse direction this takes a block number to a representative of the block.

`BlocksImage(G, P)`
`BlocksImage(G, P)`
`BlocksImage(G, P)`
`BlocksImage(G, P)`

Given a transitive permutation group G with natural G-set X and a G-invariant partition P of X, construct the group induced by the action of G on the blocks of P. In the second form, P is specified by giving a single block of the partition.

`BlocksKernel(G, P)`
`BlocksKernel(G, P)`
`BlocksKernel(G, P)`
`BlocksKernel(G, P)`

Given a transitive permutation group G with natural G-set X and a G-invariant partition P of X, construct the kernel of the action of G on the blocks of P. In the second form, P is specified by giving a single block of the partition.

Example H58E27

An imprimitive group of degree 100 constructed by Capel has several different block systems.

```
> G := sub< Sym(100) |
>   (1,21,41,61,81)(2,82,62,42,22)(3,23,43,63,83)(4,84,64,44,24)
>     (5,25,45,65,85)(6,86,66,46,26)(7,27,47,67,87)(8,88,68,48,28)
>     (9,29,49,69,89)(10,90,70,50,30)(11,31,51,71,91)(12,92,72,52,32)
>     (13,33,53,73,93)(14,94,74,54,34)(15,35,55,75,95)(16,96,76,56,36)
>     (17,37,57,77,97)(18,98,78,58,38)(19,39,59,79,99)(20,100,80,60,40),
>   (1,4,6,7,10)(2,3,5,8,9)(11,19,17,15,14)(12,20,18,16,13)(21,24,26,27,30)
>      (22,23,25,28,29)(31,39,37,35,34)(32,40,38,36,33)(41,44,46,47,50)
>      (42,43,45,48,49)(51,59,57,55,54)(52,60,58,56,53)(61,64,66,67,70)
>      (62,63,65,68,69)(71,79,77,75,74)(72,80,78,76,73)(81,84,86,87,90)
>      (82,83,85,88,89)(91,99,97,95,94)(92,100,98,96,93),
>   (1,11,2,12)(3,13,4,14)(5,16,6,15)(7,17,8,18)(9,20,10,19)(21,31,22,32)
>      (23,33,24,34)(25,36,26,35)(27,37,28,38)(29,40,30,39)(41,51,42,52)
>      (43,53,44,54)(45,56,46,55)(47,57,48,58)(49,60,50,59)(61,71,62,72)
>      (63,73,64,74)(65,76,66,75)(67,77,68,78)(69,80,70,79)(81,91,82,92)
>      (83,93,84,94)(85,96,86,95)(87,97,88,98)(89,100,90,99) >;
> MaxPart := MaximalPartition(G);
> #MaxPart;
2
> MaxPart;
GSet{@
    { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 31, 32, 33, 34, 35,
    36, 37, 38, 39, 40, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 71,
    72, 73, 74, 75, 76, 77, 78, 79, 80, 91, 92, 93, 94, 95, 96, 97,
    98, 99, 100 },
    { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 21, 22, 23, 24, 25, 26, 27, 28,
    29, 30, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 61, 62, 63, 64,
    65, 66, 67, 68, 69, 70, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90 }
@}
> MinPart := MinimalPartition(G);
> #MinPart;
50
```

We see that the group has a (maximal) system of imprimitivity consisting of 2 blocks of size 50 and a (minimal) system of imprimitivity consisting of 50 blocks of size 2.

```
> Parts := MinimalPartitions(G);
> [ #p : p in Parts ];
[ 50, 50, 50, 50, 20, 50 ]
```

Thus the group has six distinct minimal G-invariant partitions. Of these five have 50 blocks of size two while the remaining one has 20 blocks of size 5. We examine the action of G on one of the partitions into 50 blocks of size 2.

```
> f, Im, Ker := BlocksAction(G, Parts[1]);
> f;
Mapping from: GrpPerm: G to GrpPerm: Im
```

```
> Im;
Permutation group Im acting on a set of cardinality 50
Order = 7812500 = 2^2 * 5^9
    (1, 11, 31, 32, 12)(2, 13, 33, 34, 14)(3, 15, 35, 36, 16)
         (4, 17, 37, 38, 18) (5, 19, 39, 40, 20)(6, 21, 41, 42, 22)
         (7, 23, 43, 44, 24)(8, 25, 45, 46, 26)(9, 27, 47, 48, 28)
         (10, 29, 49, 50, 30)
    (1, 2, 3, 4, 5)(6, 10, 9, 8, 7)(11, 14, 16, 17, 20)(12, 13, 15, 18, 19)
         (21, 29, 27, 25, 24)(22, 30, 28, 26, 23)(31, 34, 36, 37, 40)
         (32, 33, 35, 38, 39)(41, 49, 47, 45, 44)(42, 50, 48, 46, 43)
    (1, 6)(2, 7)(3, 8)(4, 9)(5, 10)(11, 21, 12, 22)(13, 23, 14, 24)
        (15, 26, 16, 25)(17, 27, 18, 28)(19, 30, 20, 29)(31, 41, 32, 42)
        (33, 43, 34, 44)(35, 46, 36, 45)(37, 47, 38, 48)(39, 50, 40, 49)
> Ker;
Permutation group Ker acting on a set of cardinality 100
Order = 1
```

Thus, G acts faithfully on this block system.

Example H58E28

When analyzing a permutation group, it is sometimes necessary to reduce it to its primitive components. This can be done by using the constituent homomorphism and blocks homomorphism functions. We illustrate their use by analyzing the group of Rubik's cube, represented as a permutation group on 48 letters:

```
> G := sub<Sym(48) |
>     (1,3,8,6)(2,5,7,4)(9,48,15,12)(10,47,16,13)(11,46,17,14),
>     (6,15,35,26)(7,22,34,19)(8,30,33,11)(12,14,29,27)(13,21,28,20),
>     (1,12,33,41)(4,20,36,44)(6,27,38,46)(9,11,26,24)(10,19,25,18),
>     (1,24,40,17)(2,18,39,23)(3,9,38,32)(41,43,48,46)(42,45,47,44),
>     (3,43,35,14)(5,45,37,21)(8,48,40,29)(15,17,32,30)(16,23,31,22),
>     (24,27,30,43)(25,28,31,42)(26,29,32,41)(33,35,40,38)(34,37,39,36) >;
> O1 := Orbits(G);
> O1;
[
    GSet{@ 1, 3, 6, 8, 9, 11, 12, 14, 15, 17, 24, 26, 27, 29,
     30, 32, 33, 35, 38, 40, 41, 43, 46, 48 @},
    GSet{@ 2, 4, 5, 7, 10, 13, 16, 18, 19, 20, 21, 22, 23, 25, 28,
     31, 34, 36, 37, 39, 42, 44, 45, 47 @}
]
```

Thus, G has two orbits, each of size 24. We consider the restriction of the action of G to the first of these orbits.

```
> f1, Im1, Ker1 := OrbitAction(G, O1[1]);
> FactoredOrder(Im1);
[ <2, 7>, <3, 9>, <5, 1>, <7, 1> ]
> IsPrimitive(Im1);
false
```

```
> P1 := MinimalPartition(Im1);
> #P1;
8
> f2, Im2, Ker2 := BlocksAction(Im1, P1);
> FactoredOrder(Im2);
[ <2, 7>, <3, 2>, <5, 1>, <7, 1> ]
> IsPrimitive(Im2);
true
> IsSymmetric(Im2);
true
> FactoredOrder(Ker2);
[ <3, 7> ]
> IsElementaryAbelian(Ker2);
true
```

Hence the group obtained by restricting G to its first orbit is isomorphic to $Sym(8)$ acting on an elementary abelian normal subgroup of order 3^7. We next investigate the kernel $Ker1$ of the restriction of G to the first orbit of length 24. We know that $Ker1$ must fix all the points in the first orbit of G so we first take its restriction to the second orbit.

```
> f3, Im3, Ker3 := OrbitAction(Ker1, O1[2]);
> IsTransitive(Im3);
true
> FactoredOrder(Im3);
[ <2, 20>, <3, 5>, <5, 2>, <7, 1>, <11, 1> ]
> FactoredOrder(Ker3);
[]
> IsPrimitive(Im3);
false
```

The kernel acts transitively and faithfully on the second orbit. As it is imprimitive, we look at its action on a system of imprimitivity.

```
> P := MinimalPartition(Im3);
> f4, Im4, Ker4 := BlocksAction(Im3, P);
> Im4;
Permutation group Im4 acting on a set of cardinality 12
Order = 239500800 = 2^9 * 3^5 * 5^2 * 7 * 11
    (10, 12, 11)
    (9, 12, 11)
    (8, 12, 9)
    (7, 9)(8, 12)
    (6, 9, 10)
    (5, 6, 9)
    (4, 6, 9)
    (3, 6, 9)
    (2, 6, 9, 5)(4, 10)
    (1, 9, 6, 5)(4, 10)
> IsPrimitive(Im4);
true
```

```
> IsAlternating(Im4);
true
> FactoredOrder(Ker4);
[ <2, 11> ]
> IsElementaryAbelian(Ker4);
true
```

The kernel of the restriction of G to the first orbit is isomorphic to $Alt(12)$ acting on an elementary abelian group of order 2^{11}. So we now know the composition factors of G together with an indication of how they fit together.

58.10.7 Action on a Coset Space

`CosetAction(G, H: parameters)`

Given a subgroup H of the group G, construct the permutation representation of G given by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The natural homomorphism $f : G \to L$;

(b) The induced group L;

(c) The kernel K of the action (a subgroup of G).

Note that G may be any type of group. If G is a finitely presented group, then K may be returned undefined.

`Al` MonStgElt *Default :* "*Default*"

`Al := "Wang"`: Construct the coset action using Wang da Fang's algorithm which builds the action up the stabilizer chains of G and H, using a sequence of induction and block image operations. This algorithm is particularly efficient when H has fixed points that are not fixed points of G, and is the default choice when H is trivial.

`Al := "Canonical"`: Compute the cosets using an orbit algorithm, which describes each coset found by computing a canonical element of that coset. The canonical element is one with the minimal base image in the group G. The algorithm is due to Richardson, [Ric73].

`Al := "Default"`: Choose one of the above to use.

`CosetImage(G, H: parameters)`

Given a subgroup H of the group G, construct the image L of G given by the action of G on the set of (right) cosets of H in G. L is returned as a permutation group. Possible parameters are as for the previous function.

`CosetKernel(G, H)`

Given a subgroup H of the group G, construct the kernel of the action of G on the set of (right) cosets of H in G.

58.10.8 Reduced Permutation Actions

If a permutation group is intransitive or imprimitive, then orbit actions and blocks actions provide natural permutation representations of lower degree.

`TransitiveQuotient(G)`

Returns the transitive constituent of G acting on its longest orbit, together with the action homomorphism and the kernel of the action. If G is transitive then the return values are G, the identity map on G, and the trivial subgroup of G.

`PrimitiveQuotient(G)`

For a transitive group G, returns the blocks image of G acting on a maximal block system, together with the action homomorphism and the kernel of the action. If G is primitive then the return values are G, the identity map on G, and the trivial subgroup of G.

`DegreeReduction(G)`

Use a combination of orbit images and blocks images to attempt to find a faithful permutation representation of G with lower degree than G. The second return value is the isomorphism from G to the representation found. If no lower degree faithful representation is found then G and the identity map on G is returned.

58.10.9 The Jellyfish Algorithm

The Jellyfish reduction algorithm was introduced in [LNPS06]. See this article for an explanation of the name "Jellyfish". It attempts to find faithful low degree permutation representations for a family of large-base primitive permutation groups. We now define the target family as given in [LNPS06]. Consider the group $W = S_n \wr S_r$, as a permutation group in its primitive action on the set of r-tuples of k-subsets of the chosen n-set (see `PrimitiveWreathProduct`). Let M be the socle of W, $M = A_n^r$. Let G be any subgroup of W, with $M \leq G$, such that the conjugation action of G on the r copies of A_n in M is transitive. A group T is in the target family of the algorithm if T is permutation isomorphic to some such G, having $n > 2rk^2$, and $rk > 1$. The degree of such a T is $\binom{n}{k}^r$, and any such T is primitive. The image sought by the algorithm has degree nk. The utility of the algorithm is that any primitive group not in the target family is either alternating, symmetric, or has a short base.

The algorithm is one-sided Monte-Carlo in that, if it reports success, then it has found a faithful representation of the group. There is a small probability that the algorithm will find no faithful representation, even when the group given is in the target family.

Note that the Jellyfish algorithm implemented in Magma may succeed even when the input group is not in the target family. In all cases, success of the algorithm guarantees a faithful representation of the group.

The Magma implementation offers functions for testing the group for applicability of the Jellyfish algorithm. If successful, this test constructs data structures as in the cited article for quick evaluation of the homomorphism to the low-degree representation found and stores these with the group. There are also functions to compute the image and

preimage of elements under the representation map. The preimage function is an addition to the algorithms given in [LNPS06], using an extension of their data structure. The preimage algorithm is nearly linear in the degree of the large degree primitive group.

`JellyfishConstruction(G: parameters)`

`Limit` RngIntElt *Default :*

Attempt to construct a set of jellyfish for G. If unsuccessful, return false. Otherwise, construct data structures corresponding to T1 and T5 of [LNPS06], attach them to G, and return `true`. The parameter `Limit` controls how many attempts are made to find the orbits of the point stabilizer of G, which is the initial step in constructing a jellyfish. This construction phase terminates after a sequence of `Limit` random elements of G fails to change the orbits found. If the orbits found so far are not the orbits of a point stabilizer in G, the algorithm may fail. The same limit is used in the next phase, constructing the first jellyfish. The current default is the maximum of 15 and $2\lfloor \log_2 d \rfloor$, where d is the degree of G.

`JellyfishImage(G)`

If the `JellyfishConstruction` function applied to G has returned `true`, return the faithful image of G found by the jellyfish algorithm. Otherwise an error results. If the `JellyfishConstruction` has not yet been applied to G, then it is applied first with default parameters. A failure here results in an error.

`JellyfishImage(G, x)`

If the `JellyfishConstruction` function applied to G has returned `true`, return the image of x as a permutation of the jellyfish. The function will attempt this for any x in the same symmetric group as G. The algorithm may fail when $x \notin G$, in which case an error results, and this proves that x is not in G. It is possible for this map to succeed when x is not in G. In recognition of this, the parent of the result will be a symmetric group. If the `JellyfishConstruction` has not yet been applied to G, then it is applied first with default parameters. A failure here results in an error.

`JellyfishPreimage(G, x)`

If the `JellyfishConstruction` function applied to G has returned `true`, return the preimage of x as a permutation in the symmetric group of G. The element x is assumed to be a permutation of the jellyfish for G. The function will attempt this for any x in the same symmetric group as the `JellyfishImage` of G. The algorithm may fail when x is not in the image group, in which case an error results. It is possible for this map to succeed when x is not in the image group. In recognition of this, the parent of the result will be a symmetric group. If the `JellyfishConstruction` has not yet been applied to G, then it is applied first with default parameters. A failure here results in an error.

58.11 Normal and Subnormal Subgroups

58.11.1 Characteristic Subgroups and Normal Series

`DerivedSeries(G)`

The derived series of the group G. The series is returned as a sequence of subgroups. The algorithm used is described in [BC82].

`CompositionSeries(G)`

A composition series of the group G, ie. a descending chain of normal subgroups, such that each quotient is a simple group. The series is returned as a sequence of subgroups.

`CommutatorSubgroup(G)`
`DerivedSubgroup(G)`
`DerivedGroup(G)`

The derived subgroup of the group G.

`SolubleResidual(G)`
`SolvableResidual(G)`

The solvable residual (the last term of the derived series) of the group G.

`DerivedLength(G)`

The derived length of G. If G is non-soluble, the function returns the number of terms in the series terminating with the soluble residual.

`LowerCentralSeries(G)`

The lower central series of G. The series is returned as a sequence of subgroups, the first of which is the group G. The algorithm used is described in [BC82].

`NilpotencyClass(G)`

The nilpotency class of the group G. If the group is not nilpotent, the value -1 is returned.

`UpperCentralSeries(G)`

The upper central series of G. The series is returned as a sequence of subgroups commencing with the trivial subgroup. The algorithm used is to compute the centre of G and then section centralisers (see [Luk93]) up the chain. This requires computing cores of subgroups, so this function is more restricted in its range of application than `DerivedSeries` and `LowerCentralSeries`.

`Centre(G)`
`Center(G)`

Construct the centre of the group G. The centre is found by applying the function `CentralizerOfNormalSubgroup` to G in G.

`Hypercentre(G)`
`Hypercenter(G)`

Construct the hypercentre of the group G (the stationary term of the upper central series).

`pCore(G, p)`

Given a group G and a prime p, construct the maximal normal p-subgroup of G. The algorithm employed is described in Unger [Ung06b].

`pCoreQuotient(G, p)`

Given a group G and a prime p, construct the quotient of G by `K := pCore(G, p)`. The return values are the quotient, Q, represented as a permutation group of the same degree as G, the quotient map from G onto Q, and K.

`FittingSubgroup(G)`

The Fitting subgroup of the group G. It is computed as the product of the p-cores of the radical of G.

`FrattiniSubgroup(G)`

Given a group G, return the Frattini subgroup. For p-groups this is computed as the derived group with pth powers of the generators added. Solvable groups are converted to their `GrpPC` representation and the problem solved there. Non-solvable groups are treated by finding their maximal subgroups and forming the intersection, so are subject to the same restrictions as the `MaximalSubgroups` command.

`JenningsSeries(G)`

Given a p-group G, return the Jennings series for G. The series is returned as a sequence of subgroups.

`pCentralSeries(G, p)`

Given a soluble group G, and a prime p dividing $|G|$, return the lower p-central series for G. The series is returned as a sequence of subgroups.

`SubnormalSeries(G, H)`

Given a group G and a subnormal subgroup H of G, return a sequence of subgroups commencing with G and terminating with H, such that each subgroup is normal in the previous one. If H is not subnormal in G, the empty sequence is returned.

Example H58E29

We compute the various series in the wreath product of the symmetric group of degree 4 with the dihedral group of order 8 (a soluble group).

```
> G := WreathProduct(Sym(4), DihedralGroup(4));
> G;
Permutation group G acting on a set of cardinality 16
    (1, 5, 9, 13)(2, 6, 10, 14)(3, 7, 11, 15)(4, 8, 12, 16)
    (1, 13)(2, 14)(3, 15)(4, 16)(5, 9)(6, 10)(7, 11)(8, 12)
    (1, 2, 3, 4)
    (1, 2)
> [ FactoredOrder(H) : H in DerivedSeries(G) ];
[
    [ <2, 15>, <3, 4> ],
    [ <2, 12>, <3, 4> ],
    [ <2, 9>, <3, 4> ],
    [ <2, 8>, <3, 4> ],
    [ <2, 8> ],
    []
]
> DerivedLength(G);
5
> [ FactoredOrder(H) : H in LowerCentralSeries(G) ];
[
    [ <2, 15>, <3, 4> ],
    [ <2, 12>, <3, 4> ],
    [ <2, 10>, <3, 4> ],
    [ <2, 9>, <3, 4> ],
    [ <2, 8>, <3, 4> ]
]
> NilpotencyClass(G);
-1
> Centre(G);
Permutation group acting on a set of cardinality 16
Order = 1
    Id($)
> pCentralSeries(G, 2);
[
    [ <2, 15>, <3, 4> ],
    [ <2, 12>, <3, 4> ],
    [ <2, 10>, <3, 4> ],
    [ <2, 9>, <3, 4> ],
    [ <2, 8>, <3, 4> ]
]
> [ FactoredOrder(H) : H in pCentralSeries(G, 3) ];
[
    [ <2, 15>, <3, 4> ]
```

```
]
```

58.11.2 Maximal and Minimal Normal Subgroups

`MaximalNormalSubgroup(G)`

A maximal normal subgroup of G. The trivial subgroup is returned if G is simple. The algorithm takes homomorphic reductions to a primitive group and then uses O'Nan-Scott type considerations to get its result.

`MinimalNormalSubgroups(G)`

The minimal normal subgroups of G. These are obtained by first computing the socle of G and then splitting off the normal factors.

58.11.3 Lattice of Normal Subgroups

`NormalSubgroups(G)`

The normal subgroups of G. These are determined by the method of Cannon and Souvignier [CS].

`NormalLattice(G)`

The normal subgroup lattice of G. The subgroups are first found using the same algorithm as the function `NormalSubgroups` and then inclusions are determined.

Example H58E30

We determine all normal subgroups of the wreath product of $Sym(8)$ and the dihedral group of order 8.

```
> G := WreathProduct(Sym(8), DihedralGroup(4));
> Order(G);
21143266346926080000
> time N := NormalSubgroups(G);
Time: 1.050
> #N;
29
> [ < Order(H`subgroup), FactoredOrder(H`subgroup) > : H in N ];
[
    <1, []>,
    <165181768335360000, [ <2, 24>, <3, 8>, <5, 4>, <7, 4> ]>,
    <330363536670720000, [ <2, 25>, <3, 8>, <5, 4>, <7, 4> ]>,
    <660727073341440000, [ <2, 26>, <3, 8>, <5, 4>, <7, 4> ]>,
    <1321454146682880000, [ <2, 27>, <3, 8>, <5, 4>, <7, 4> ]>,
    <1321454146682880000, [ <2, 27>, <3, 8>, <5, 4>, <7, 4> ]>,
    <1321454146682880000, [ <2, 27>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
```

```
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <21143266346926080000, [ <2, 31>, <3, 8>, <5, 4>, <7, 4> ]>
]
```

58.11.4 Composition and Chief Series

`ChiefFactors(G)`

Given a group G, return a sequence of the isomorphism types $< f, d, q, m >$ of the chief factors. An isomorphism type in a chief factor should be understood as the direct product of m copies of the simple group described by $< f, d, q >$ (see `CompositionFactors` below). For the algorithm, see Unger [Ung].

`ChiefSeries(G)`

Given a group G, return the chief series of G and a sequence of the corresponding isomorphism types $< f, d, q, m >$ of the chief factors. An isomorphism type in a chief factor should be understood as the direct product of m copies of the simple group described by $< f, d, q >$ (see `CompositionFactors` below). The series will be organised to include the soluble radical of G, and, if G is insoluble, the socle of the quotient of G by the soluble radical.

f	Family name
1	A(d, q)
2	B(d, q)
3	C(d, q)
4	D(d, q)
5	G$(2, q)$
6	F$(4, q)$
7	E$(6, q)$
8	E$(7, q)$
9	E$(8, q)$
10	2A(d, q)
11	2B$(2, q)$
12	2D(d, q)
13	3D$(4, q)$
14	2G$(2, q)$
15	2F$(4, q)$
16	2E$(6, q)$
17	Alternating(d)
18	Sporadic group — see Table 2.
19	Cyclic(q)

Table 1: Family numbers and names

d	Group name
1	M_{11}
2	M_{12}
3	M_{22}
4	M_{23}
5	M_{24}
6	J_1
7	HS
8	J_2
9	MCL
10	SUZ
11	J_3
12	CO_1
13	CO_2
14	CO_3
15	HE
16	$M(22)$
17	$M(23)$
18	$M(24)$
19	LY
20	RU
21	ON
22	TH
23	HA
24	BM
25	M
26	J_4

Table 2: Sporadic groups

`CompositionFactors(G)`

Given a permutation group G, return a sequence S of tuples that represent the composition factors of G, ordered according to some composition series of G. Each tuple is a triple of integers f, d, q that defines the isomorphism type of the corresponding composition factor. A triple $\langle f, d, q\rangle$ describes a simple group as follows. The integer f defines the family to which the group belongs, and d and q are the parameters of the family. The length of the sequence S is the number of composition factors of G. The algorithm used is the "tabular" algorithm of Kantor [Kan91], extended to be valid for groups of degree $\leq 10^7$. The families are listed in Tables 1 and 2 on page 1590.

Example H58E31

We illustrate the function `CompositionFactors` by applying it to the group associated with Rubik's cube.

```
> G := sub<Sym(48) |
>     (1,3,8,6)(2,5,7,4)(9,48,15,12)(10,47,16,13)(11,46,17,14),
>     (6,15,35,26)(7,22,34,19)(8,30,33,11)(12,14,29,27)(13,21,28,20),
>     (1,12,33,41)(4,20,36,44)(6,27,38,46)(9,11,26,24)(10,19,25,18),
>     (1,24,40,17)(2,18,39,23)(3,9,38,32)(41,43,48,46)(42,45,47,44),
>     (3,43,35,14)(5,45,37,21)(8,48,40,29)(15,17,32,30)(16,23,31,22),
>     (24,27,30,43)(25,28,31,42)(26,29,32,41)(33,35,40,38)(34,37,39,36)
>         >;
> FactoredOrder(G);
[ <2, 27>, <3, 14>, <5, 3>, <7, 2>, <11, 1> ]
> CompositionFactors(G);
    G
    |  Cyclic(2)
    *
    |  Alternating(12)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Alternating(8)
    *
    |  Cyclic(3)
    *
    |  Cyclic(3)
    *
```

```
|  Cyclic(3)
*
|  Cyclic(3)
*
|  Cyclic(3)
*
|  Cyclic(3)
*
|  Cyclic(3)
1
```

58.11.5 The Socle

`Socle(G)`

The socle of the group G. This is computed using the algorithms described in Cannon and Holt [CH97].

`SocleFactor(G)`

A simple factor of the socle of the group G.

`SocleFactors(G)`

The simple factors of the socle of the group G. The index of each factor in the sequence corresponds to the points of the image group of `SocleAction` and `SocleImage`.

`SocleSeries(G)`

A chain of subgroups

$$S_1, S_1 \times S_2, \ldots, S_1 \times \ldots \times S_r,$$

where $S_1, ..., S_r$ are the simple factors of the socle of the primitive group G.

`EARNS(G)`

The elementary abelian regular normal subgroup (EARNS) of the primitive group G. If G does not have an EARNS, then the trivial subgroup is returned. The algorithm used is that of Neumann [Neu86].

`IsAffine(G)`

Decide if the permutation group G is of primitive affine type. If so, the elementary abelian regular normal subgroup of G is returned as second return value. If the group G is either intransitive or transitive and imprimitive or primitive and not of affine type, then the result will be false (only). This function combines `IsTransitive`, `IsPrimitive` and `EARNS`.

`AffineAction(G)`

Given a primitive group G which has a non-trivial elementary abelian regular normal subgroup A, construct the representation of G given by the action of G on elements of the elementary abelian group A. The image is realised as a point-stabilizer in G and the kernel of the action is A. As with the other action functions, `AffineAction` returns the homomorphism, the image and the kernel of the action.

`AffineImage(G)`

Given a primitive group G which has an elementary abelian regular normal subgroup A, construct the permutation group that results from the action of G on elements of the elementary abelian group A. This image is realised as a point-stabilizer in G.

`AffineKernel(G)`

Given a primitive group G which has a non-trivial elementary abelian regular normal subgroup A, construct the kernel of the action of G on elements of the elementary abelian group A. This kernel equals A.

`SocleAction(G)`

Given a non-trivial permutation group G which has trivial Fitting subgroup, construct the permutation representation of G given by the action of G on the simple factors of N. Note that a primitive group has a perfect socle if and only if it has no elementary abelian regular normal subgroup. As with the other action functions, `SocleAction` returns the homomorphism, the image and the kernel of the action. The socle factor corresponding to point i in the support of the image group is the ith element in the sequence `SocleFactors(G)`.

`SocleImage(G)`

Given a non-trivial permutation group G which has trivial Fitting subgroup, construct the permutation group L induced by the action of G on the simple factors of N.

`SocleKernel(G)`

Given a non-trivial permutation group G which has trivial Fitting subgroup, construct the kernel of the action of G on the simple factors of N.

`SocleQuotient(G)`

Given a permutation group G which has trivial Fitting subgroup, construct a permutation representation of G/N. If U_i denote the simple factors of N, then the degree of the result is bounded by $\sum_i |Out(U_i)|$ (see Cannon and Souvignier [CS]). Note that a primitive group has a perfect socle if and only if it has no elementary abelian regular normal subgroup. `SocleQuotient` returns G/N, the quotient homomorphism and the kernel of the map (which is the socle of G).

`RefineSection(G, M, N)`

Given M, N normal subgroups of G with $N < M$, return a sequence of G-normal subgroups $L_1, \ldots, L_r$ with $N = L_0$, $L_i < L_{i+1}$ and $L_r = M$ such that each of the quotients L_{i+1}/L_i is either elementary abelian or a direct product of non-abelian simple groups.

Example H58E32

We examine the normal structure of a primitive group, the primitive-wreath product of Sym(5) and Sym(3) (with product action).

```
> G := PrimitiveWreathProduct(Sym(5), Sym(3));
> FactoredOrder(G);
[ <2, 10>, <3, 4>, <5, 3> ]
> E := EARNS(G);
> E;
Permutation group E acting on a set of cardinality 125
Order = 1
> DerivedSeries(G);
[
    Permutation group G acting on a set of cardinality 125
    Order = 10368000 = 2^10 * 3^4 * 5^3
    Permutation group acting on a set of cardinality 125
    Order = 2592000 = 2^8 * 3^4 * 5^3,
    Permutation group acting on a set of cardinality 125
    Order = 864000 = 2^8 * 3^3 * 5^3,
    Permutation group S acting on a set of cardinality 125
    Order = 216000 = 2^6 * 3^3 * 5^3
]
> S := Socle(G);
> S;
Permutation group S acting on a set of cardinality 125
Order = 216000 = 2^6 * 3^3 * 5^3
> Q := SocleFactors(G);
> Q;
[
    Permutation group acting on a set of cardinality 125
    Order = 60 = 2^2 * 3 * 5,
    Permutation group acting on a set of cardinality 125
    Order = 60 = 2^2 * 3 * 5,
    Permutation group acting on a set of cardinality 125
    Order = 60 = 2^2 * 3 * 5
]
> R := SocleSeries(G);
> R;
[
    Permutation group acting on a set of cardinality 125
    Order = 60 = 2^2 * 3 * 5,
```

```
    Permutation group acting on a set of cardinality 125
    Order = 3600 = 2^4 * 3^2 * 5^2,
    Permutation group acting on a set of cardinality 125
    Order = 216000 = 2^6 * 3^3 * 5^3
]
> SQ := SocleQuotient(G);
> SQ;
Permutation group SQ acting on a set of cardinality 6
Order = 48 = 2^4 * 3
    (1, 2, 3)(4, 5, 6)
    (2, 3)(4, 5)
    Id($)
    (2, 4)
```

58.11.6 The Soluble Radical and its Quotient

Very efficient algorithms have been developed for computing invariants such as subgroups, normal subgroups and conjugacy classes of elements for soluble groups defined by means of polycyclic presentations. Almost all such algorithms employ a top-down *Lifting Strategy*. Let P be a quotient-invariant property for a soluble group. In general, an algorithm that constructs the set of elements or subgroups $X_P(G)$ satisfying property P for the group G, proceeds as follows: Let G be a non-simple soluble group and let N be a normal subgroup of G. The set $X_P(G/N)$ is constructed and its elements are lifted back into G, thereby yielding $X_P(G)$. This process is usually iterated with successive normal subgroups N being chosen as the terms of some descending normal series (e.g., an elementary abelian series).

In generalizing this approach to permutation groups, our approach has been to construct the soluble radical R of G, use special methods to solve the problem for the quotient G/R, and then proceed (as in the case of a soluble group) to lift the solution down the successive terms of an elementary abelian series for G using the Lifting Strategy. Derek Holt has shown that the quotient group G/R has a faithful permutation representation of degree no greater than that of G.

The functions in this section enable the user to construct the radical, its quotient and an elementary abelian series.

`Radical(G)`
`SolubleRadical(G)`
`SolvableRadical(G)`

Given a group G, return the maximal normal solvable subgroup of G. The algorithm used is described in Unger [Ung06b].

`RadicalQuotient(G)`

Given a group G, compute a representation of the quotient G/R where R is the (solvable) radical of G. The resulting representation has the same degree as G. Both the permutation group Q isomorphic to G/R and a homomorphism $\phi : G \to Q$ are returned. The algorithm proceeds by repeatedly applying `AbelianNormalQuotient` up the terms of the derived series of the radical. The third return value is R, the radical of G and the kernel of the homomorphism.

`ElementaryAbelianSeries(G: parameters)`

`ElementaryAbelianSeries(G, N: parameters)`

`LayerSizes` SeqEnum[RngIntElt] *Default :* []

An elementary abelian series is a chain of normal subgroups $R = N_1 > N_2 > ... > N_r = 1$ with the property that the quotient of each pair of successive terms in the series is elementary abelian and that there is no group $R < H < G$ such that H/R is elementary abelian and H normal in G. The top of the series R is called the solvable radical and is the maximal normal solvable subgroup of G.

In the second form N must be a normal subgroup of G and the returned series has the form $R = N_1 > N_2 > ... > N_r = N$, so is an elementary abelian series for G/N.

The parameter `LayerSizes` controls possible refinement of the series. The default is no refinement. As an example, take `LayerSizes := [ 2, 5, 3, 4, 7, 3, 11, 2, 17, 1]`. When constructing an elementary abelian series for the group, attempt to split 2-layers of size gt 2^5, 3-layers of size gt 3^4, etc. The implied exponent for 13 is 2 and for all primes greater than 17 the exponent is 1. Setting `LayerSizes` to `[2, 1]` will attempt to split all layers, resulting in a portion of a chief series for G.

`ElementaryAbelianSeriesCanonical(G)`

Gives a similar result to using `ElementaryAbelianSeries`, except the series returned depends only on the isomorphism type of the solvable radical, and consists of characteristic subgroups of G. This function may be slower than `ElementaryAbelianSeries`.

Example H58E33

We illustrate these functions by considering the group of degree 16 generated by the following permutations:

$$(1, 8, 11, 3, 6, 14, 15, 10)(2, 7, 12, 4, 5, 13, 16, 9),$$

$$(1, 2)(3, 16, 9, 14, 8, 12)(4, 15, 10, 13, 7, 11),$$

$$(1, 13, 12, 16)(2, 14, 11, 15)(7, 9)(8, 10),$$

```
> G := PermutationGroup< 16 |
>         (1, 8, 11, 3, 6, 14, 15, 10)(2, 7, 12, 4, 5, 13, 16, 9),
>         (1, 2)(3, 16, 9, 14, 8, 12)(4, 15, 10, 13, 7, 11),
```

```
>         (1, 13, 12, 16)(2, 14, 11, 15)(7, 9)(8, 10) >;
> Radical(G);
Permutation group acting on a set of cardinality 16
Order = 256 = 2^8
   (3, 4)(5, 6)(7, 8)(13, 14)(15, 16)
   (3, 4)(7, 8)(9, 10)(11, 12)
   (7, 8)(13, 14)
   (1, 2)(7, 8)(9, 10)(11, 12)(13, 14)(15, 16)
   (9, 10)
   (15, 16)
   (11, 12)(15, 16)
   (13, 14)(15, 16)
> RadicalQuotient(G);
Permutation group acting on a set of cardinality 16
Order = 40320 = 2^7 * 3^2 * 5 * 7
   (1, 7, 11, 3, 5, 13, 15, 9)(2, 8, 12, 4, 6, 14, 16, 10)
   (3, 15, 9, 13, 7, 11)(4, 16, 10, 14, 8, 12)
   (1, 13, 11, 15)(2, 14, 12, 16)(7, 9)(8, 10)
Mapping from: GrpPerm: g to GrpPerm: $, Degree 16
> ElementaryAbelianSeries(G);
[
   Permutation group acting on a set of cardinality 16
   Order = 256 = 2^8
       (3, 4)(5, 6)(7, 8)(13, 14)(15, 16)
       (3, 4)(7, 8)(9, 10)(11, 12)
       (7, 8)(13, 14)
       (1, 2)(7, 8)(9, 10)(11, 12)(13, 14)(15, 16)
       (9, 10)
       (15, 16)
       (11, 12)(15, 16)
       (13, 14)(15, 16),
   Permutation group acting on a set of cardinality 16
   Order = 1
]
```

58.11.7 Complements and Supplements

`Complements(G, M)`

Given a group G and a normal subgroup M, this function returns a sequence containing one representative from each conjugacy class of complements of M in G.

`Complements(G, M, N)`

Given a group G, a normal subgroup M of G and a normal subgroup N of G, that is strictly contained in M, the function returns a sequence comprising representatives for the conjugacy classes of complements of M/N in G/N, as subgroups of G.

`HasComplement(G, M)`

The group M must be a normal subgroup of G. Returns whether M has a complement in G and, if so, one such complement.

`Supplements(G, M)`

Given a group G and a soluble normal subgroup M of G, the function returns a sequence containing one representative from each conjugacy class of minimal supplements for M in G.

`Supplements(G, M, N)`

Given a group G, a normal subgroup M of G and a normal subgroup N of G such that (a), N is strictly contained in M, and (b), M/N is soluble, the function returns a sequence comprising representatives for the conjugacy classes of minimal supplements of M/N in G/N, as subgroups of G.

`HasSupplement(G, M)`

The group M must be a soluble normal subgroup of G. Returns whether M has a proper supplement in G and, if so, one such supplement.

Example H58E34

We illustrate these functions by considering a normal subgroup H of the group G of degree 16 generated by the following permutations:

$$(1,3,2,4)(5,16,6,13)(7,14,8,15)(9,12,11,10),$$
$$(1,16,9)(2,15,12)(3,14,11)(4,13,10)(6,8,7).$$

```
> G := PermutationGroup< 16 |
>         (1, 3, 2, 4)(5, 16, 6, 13)(7, 14, 8, 15)(9, 12, 11, 10),
>         (1, 16, 9)(2, 15, 12)(3, 14, 11)(4, 13, 10)(6, 8, 7) >;
Permutation group G acting on a set of cardinality 16
Order = 165888 = 2^11 * 3^4
    (1, 3, 2, 4)(5, 16, 6, 13)(7, 14, 8, 15)(9, 12, 11, 10)
    (1, 16, 9)(2, 15, 12)(3, 14, 11)(4, 13, 10)(6, 8, 7)
> H := ncl< G | (6, 7, 8)(14, 16, 15) >;
> H;
Permutation group H acting on a set of cardinality 16
Order = 6912 = 2^8 * 3^3
   (6, 7, 8)(14, 16, 15)
   (6, 7, 8)(13, 14, 15)
   (6, 7, 8)(9, 12, 11)
   (5, 8, 7)(13, 14, 15)
   (6, 7, 8)(10, 11, 12)
   (1, 2, 3)(6, 7, 8)
   (2, 4, 3)(6, 7, 8)
```

```
> C := Complements(G, H);
> C;
[
   Permutation group acting on a set of cardinality 16
   Order = 24 = 2^3 * 3
       (3, 4)(5, 14)(6, 15)(7, 16)(8, 13)(10, 12)
       (2, 4)(6, 7)(9, 14)(10, 15)(11, 13)(12, 16)
       (1, 14)(2, 15)(3, 16)(4, 13)(7, 8)(10, 11)
       (1, 14, 9)(2, 13, 10)(3, 16, 12)(4, 15, 11)(6, 8, 7)
]
```

So the normal subgroup has one conjugacy class of complements. We check that the representative subgroup is indeed a complement for H.

```
> K := C[1];
> IsTrivial(K meet H );
true
> #K * #H eq #G;
true
```

58.11.8 Abelian Normal Subgroups

`AbelianNormalSubgroup(G)`

An abelian normal subgroup of G. If none exists, the trivial subgroup is returned.

`AbelianNormalQuotient(G, H)`

A quotient of G by an abelian normal subgroup that contains the abelian normal subgroup H. The quotient is represented as a permutation group of the same degree as G. The other values returned are the quotient epimorphism and its kernel K. The kernel K will contain H, $\#K$ and $\#H$ will have the same prime divisors, and if H is elementary abelian then so is K.

`SolubleNormalQuotient(G, H)`

A quotient of G by a soluble normal subgroup that contains the soluble normal subgroup H. The quotient is represented as a permutation group of the same degree as G. The other values returned are the quotient epimorphism and its kernel K. As with `AbelianNormalQuotient`, K will contain H, and $\#K$ and $\#H$ will have the same prime divisors.

`ElementaryAbelianNormalSubgroup(G)`

An elementary abelian normal subgroup of G. If none exists, the trivial subgroup is returned. The group returned is the last non-trivial group in an elementary abelian series for the radical of G.

`pElementaryAbelianNormalSubgroup(G, p)`

An elementary abelian normal p-subgroup of G. If none exists, the trivial subgroup is returned. The group returned is the last non-trivial group in an elementary abelian series for the p-core of G.

`MEANS(G)`

A minimal elementary abelian normal subgroup of G.

`MEANS(G, N)`

A minimal elementary abelian normal subgroup of G that lies in the elementary abelian normal subgroup N of G.

58.12 Cosets and Transversals

58.12.1 Cosets

`H * g`

Right coset of the subgroup H of the group G, where g is an element of G.

`DoubleCoset(G, H, g, K)`

The double coset $H * g * K$ of the subgroups H and K of the group G, where g is an element of G.

`DoubleCosetRepresentatives(G, H, K)`

Given a group G and two subgroups H and K of G, return a sequence S containing representatives of the H-K-double cosets in G. The first element of S is guaranteed to be the identity element of G. The second return sequence gives the sizes of the corresponding double cosets. The algorithm used refines double cosets down a chain of subgroups from G to one of H or K.

`ProcessLadder(L, G, U)`

Verbose `DoubleCosets` *Maximum :* 3

Given permutation groups $U < G$ and a sequence of permutation groups L such that $L_1 = G$, compute data for computations with the L_n-U-double cosets in G. The algorithm relies on the indices $(L_i : L_{i+1})$ (for $L_i < L_{i+1}$) or $(L_{i+1} : L_i)$ otherwise to be small. In contrast to the method used by `DoubleCosetRepresentatives`, the sequence used in the computation is a ladder, not necessarily a descending chain. For details see [Sch90].

`GetRep(p, R)`

For R being the result of a call to `ProcessLadder` and a permutation $p \in G$, return the canonical double coset representative for p.

`DeleteData(R)`

Deletes the data computed using `ProcessLadder`.

`YoungSubgroupLadder(L)`

`Full` RngIntElt *Default :* `false`

Computes a ladder from the full symmetric group down to the Young subgroup parametrised by the sequence L suitable for double coset enumeration using `ProcessLadder`. The optional parameter `Full` can be used if the Young subgroup should be considered as a subgroup of the symmetric group on `Full` points rather than on `&*L`.

`StabilizerLadder(G, d)`

Given a subgroup G of the symmetric group of degree n and a monomial in n indeterminates, compute a ladder down from the full symmetric group to the stabilizer of the monomial, suitable for processing with `ProcessLadder`.

`x in C`

Returns `true` if element g of group G lies in the coset C.

`x notin C`

Returns `true` if element g of group G does not lie in the coset C.

`C1 eq C2`

Returns `true` if the coset C_1 is equal to the coset C_2.

`C1 ne C2`

Returns `true` if the coset C_1 is not equal to the coset C_2.

`#C`

The cardinality of the coset C.

`CosetTable(G, H)`

The (right) coset table for G over subgroup H relative to its defining generators.

`#CosetTable(G, f)`

The coset table for G corresponding to the permutation representation f of G, where f is a homomorphism of G onto a transitive permutation group.

58.12.2 Transversals

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a permutation group G and a subgroup H of G, this function returns

(a) An indexed set of elements T of G forming a right transversal for G over H; and

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and $g \in G$, ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

`TransversalProcess(G, H)`

Given a permutation group G and H, a subgroup of G, create a process to run through a left transversal for H in G. The method used is a backtrack search for a canonical coset representative. `TransversalProcess` can be used when the index of H in G is too large to allow a full transversal to be created.

`TransversalProcessRemaining(P)`

The number of coset representatives the process has yet to produce. Initially this will be the index of the subgroup in the group.

`TransversalProcessNext(P)`

Advance the process to the next coset representative and return that representative. This may only be used when `TransversalProcessRemaining(P)` is positive. The first call to `TransversalProcessNext` will always give the identity element.

`ShortCosets(p, H, G)`

Computes a set of representatives for the transversal of G modulo H of all cosets that contain p. This computation does not do a full transversal of G modulo H and may therefore be used even if the index of $(G : H)$ is very large.

58.13 Presentations

In this section we describe how to compute a presentation in terms of generators and relations for a permutation group and also how to obtain a representation of a permutation as word in the defining generators.

58.13.1 Generators and Relations

`FPGroup(G)`

Construct a presentation for the permutation group G on the set of defining generators and return the presentation in the form of a finitely presented group F that is isomorphic to G. The presentation is obtained by first computing the regular representation of G and then using the Todd-Coxeter Schreier algorithm to construct a presentation on the strong generators. In this situation the strong generators are identical to the defining generators.

A group homomorphism $\phi : F \to G$, defining G as a permutation representation of F, is also returned.

`FPQuotient(G, N)`

Given a normal subgroup N of G, compute an fp-group representation F of the quotient G/N and the homomorphism $\phi : G \to F$.

`FPGroupStrong(G: parameters)`

`Random`	BoolElt	*Default :* `true`
`Run`	RngIntElt	*Default :* 20

Construct a presentation for the permutation group G on a set of strong generators and return the presentation in the form of a finitely presented group F that is isomorphic to G. In Magma, a combination of the Schreier Todd-Coxeter Sims algorithm and the Brownie-Cannon-Sims verification procedure is used to construct the presentation. See Leon [Leo80] and Gebhardt [Geb00] for more details of the individual algorithms.

If strong generators are not already known for G, they will be constructed. If strong generators have to be constructed, the parameters `Random` and `Run` may be used to control the application of the random schreier algorithm to construct a probable BSGS before commencing the construction of the presentation. If `Random` is set to false then no randomising is performed, and the algorithm becomes the straight STCS algorithm. In the case in which strong generators are already known for G, the presentation will be on these strong generators.

The presentation will have the property that it includes a presentation for each group in the stabilizer chain of the BSGS.

The group isomorphism $\phi : F \to G$, defining G as a permutation representation of F, is also returned.

58.13.2 Permutations as Words

Consider a permutation group G defined on d generators. The *word group* of G is a free group W of rank d. Then we regard G as a homomorphic image of F with associated homomorphism $\phi : W \to G$. All operations involving words in the generators of G will be performed in W.

`WordGroup(G)`

Given a permutation group G defined on d generators, return (a) a free group W on d generators represented as a group whose elements are defined by straight-line programs (SLP group), and (b) the homomorphism ϕ from W to G such that $W.i \to G.i$, for $i = 1, \ldots, d$. The group W associated with G by this function will be referred to as the *word group* for G.

`InverseWordMap(G)`

Given a permutation group G and its associated word group W with canonical homomorphism $\phi : W \to G$, construct the inverse mapping ρ. Thus, given a permutation g of G, $g@\rho$ returns an element in the preimage of g under ϕ. If the word group W does not already exist, it will be created.

`ActingWord(G, x, y)`

Given points x and y belonging to the same G-orbit of the natural G-set X, return a word w in the word group W of G such that $x^{\phi(w)} = y$. Here ϕ is the canonical homomorphism from W to G.

58.14 Automorphism Groups

The automorphism group of a permutation group may be computed in MAGMA, subject to the same restrictions on the group as when computing maximal subgroups. (That is, the non-abelian composition factors of the group must appear in a certain database.) The methods used are those described in Cannon and Holt [CH03]. Isomorphism of permutation groups may also be determined using the same methods.

`AutomorphismGroup(G: parameters)`

Compute the full automorphism group of the permutation group G.

`IsIsomorphic(G, H: parameters)`

Test whether or not the two permutation groups G and H are isomorphic as abstract groups. If so, both the result `true` and an isomorphism from G to H is returned. If not, the result `false` is returned.

Example H58E35

We take some groups of order 120 and test for isomorphism.

```
> G1 := PermutationGroup<20 |
>  [ 2, 5, 9, 11, 12, 3, 17, 13, 18, 16, 7, 15, 10, 8, 1,
>   14, 20, 19, 6, 4 ],
>  [ 3, 6, 1, 10, 14, 2, 18, 17, 15, 4, 16, 13, 12, 5, 9,
>   11, 8, 7, 20, 19 ] >;
> #G1;
120
> G2 := PermutationGroup<24 |
```

```
>   [ 2, 4, 6, 5, 1, 7, 8, 3, 13, 15, 14, 16, 11, 9, 12, 10,
>   19, 20, 18, 17, 24, 21, 22, 23 ],
>   [ 3, 1, 2, 7, 4, 9, 5, 11, 10, 6, 12, 8, 16, 15, 18, 17,
>   13, 14, 21, 23, 22, 19, 24, 20 ],
>   [ 4, 5, 7, 1, 2, 8, 3, 6, 11, 12, 9, 10, 14, 13, 16, 15,
>   18, 17, 20, 19, 23, 24, 21, 22 ] >;
> #G2;
120
> IsIsomorphic(G1, G2);
false
> flag, isom := IsIsomorphic(G1, Sym(5));
> flag;
true
> (G1.1)@ isom;
(1, 3, 5, 4, 2)
```

The reader is invited to check that `G2` is perfect while `G1` is not, so the `false` result for their isomorphism is correct. What is the automorphism group of `G2`?

```
> A := AutomorphismGroup(G2);
> #A;
120
> #Centre(G2);
2
> OuterFPGroup(A);
Finitely presented group on 1 generator
Relations
  $.1^2 = Id($)
> A.1;
Automorphism of GrpPerm: G2, Degree 24, Order 2^3 * 3 * 5
which maps:
  (1, 2, 4, 5)(3, 6, 7, 8)(9, 13, 11, 14)(10, 15, 12,
    16)(17, 19, 18, 20)(21, 24, 23, 22) |--> (1, 16, 4,
    15)(2, 21, 5, 23)(3, 20, 7, 19)(6, 11, 8, 9)(10, 24, 12,
    22)(13, 17, 14, 18)
  (1, 3, 2)(4, 7, 5)(6, 9, 10)(8, 11, 12)(13, 16, 17)(14,
    15, 18)(19, 21, 22)(20, 23, 24) |--> (1, 11, 16)(2, 18,
    19)(3, 23, 6)(4, 9, 15)(5, 17, 20)(7, 21, 8)(10, 22,
    13)(12, 24, 14)
  (1, 4)(2, 5)(3, 7)(6, 8)(9, 11)(10, 12)(13, 14)(15,
    16)(17, 18)(19, 20)(21, 23)(22, 24) |--> (1, 4)(2, 5)(3,
    7)(6, 8)(9, 11)(10, 12)(13, 14)(15, 16)(17, 18)(19,
    20)(21, 23)(22, 24)
> IsInnerAutomorphism(A.4);
false
```

So the outer automorphism group of `G2` has order 2, and `A.4` gives this automorphism.

58.15 Cohomology

In the following description, G is a finite permutation group, p is a prime number, and K is the finite field of order p. Further, F is a finitely presented group having the same number of generators as G, and is such that its relations are satisfied by the corresponding generators of G. In other words, the mapping taking the i-th generator of F to the i-th generator of G must be an epimorphism. Usually this mapping will be an isomorphism, although this is not mandatory. The algorithms used are those of Holt, see [Hol84], [Hol85a] and [Hol85b].

`pMultiplicator(G, p)`

Given the group G and a prime p, return the invariant factors of the p-part of the Schur multiplicator of G.

`pCover(G, F, p)`

Given the group G and the finitely presented group F such that G is an epimorphic image of G in the sense described above, return a presentation for the p-cover of G, constructed as an extension of the p-multiplier by F.

`CohomologicalDimension(G, M, i)`

Given the group G, the $K[G]$-module M and an integer i (equal to 1 or 2), return the dimension of the i-th cohomology group of G acting on M.

`ExtensionProcess(G, M, F)`

Create an extension process for the group G by the module M.

`Extension(P, Q)`

`#NextExtension(P)`

Return the next extension of G as defined by the process P.

Assume that F is isomorphic to the permutation group G, and that we wish to determine presentations for one or more extensions of the K-module M by F, where K is the field of p elements. We first create an extension process using `ExtensionProcess(G, M, F)`. The possible extensions of M by G are in one-one correspondence with the elements of the second cohomology group $H^2(G, M)$ of G acting on M. Let $b_1, \ldots, b_l$ be a basis of $H^2(G, M)$. A general element of $H^2(G, M)$ therefore has the form $a_1 b_1 + \cdots + a_l b_l$ and so can be defined by a sequence Q of l integers $[a_1, \ldots, a_l]$. Now, to construct the corresponding extension of M by G we call the function `Extension(P, Q)`. The required extension is returned as a finitely presented group. If all the extensions are required then they may be obtained successively by making p^l calls to the function `NextExtension`.

`SplitExtension(G, M, F)`

The split extension of the module M by the group G.

Example H58E36

We construct a presentation for A_6 over its Schur multiplicator. First we find the size of the multiplicator by applying the `pMultiplicator` function to each relevant prime.

```
> G := Alt(6);
> &cat [pMultiplicator(G, p[1]): p in FactoredOrder(G)];
[ 2, 3, 1 ]
```

The multiplicator has order $2 \times 3 = 6$. We next construct the two-fold cover of A_6. We use the `FPGroup` function to get a presentation for A_6.

```
> F := FPGroup(G);
> F2 := pCover(G, F, 2);
```

Now we construct a three-fold cover of the two-fold cover to get the extension we are after. First we need a permutation representation of `F2`, the two-fold covering group.

```
> G2 := DegreeReduction(CosetImage(F2, sub<F2|>));
> Degree(G2);
144
> #G2;
720
> F6 := pCover(G2, F2, 3);
> F6;
Finitely presented group F6 on 4 generators
Relations
  F6.4^3 = Id(F6)
  (F6.1, F6.4) = Id(F6)
  (F6.2, F6.4) = Id(F6)
  (F6.3, F6.4) = Id(F6)
  F6.3^2 = Id(F6)
  (F6.1, F6.3) = Id(F6)
  (F6.2, F6.3) = Id(F6)
  F6.1^4 * F6.3 = Id(F6)
  F6.2^3 * F6.3 = Id(F6)
  F6.1^-1 * F6.2^-1 * F6.1 * F6.2 * F6.1^-1 * F6.2^-1 *
  F6.1 * F6.2 * F6.3 * F6.4 = Id(F6)
  F6.1^-1 * F6.2 * F6.1^2 * F6.2 * F6.1^2 * F6.2 *
  F6.1^-2 * F6.2 * F6.1^-1 * F6.3 * F6.4^-1 = Id(F6)
  F6.2 * F6.1 * F6.2 * F6.1 * F6.2 * F6.1 * F6.2 *
  F6.1 * F6.2 * F6.1 * F6.3 * F6.4 = Id(F6)
> AbelianQuotientInvariants(F6);
[]
```

The group `F6` is the six-fold cover of A_6. We easily see from the presentation that the 3rd and 4th generators generate a central cyclic subgroup of order 6. The sequence of invariants for the maximal abelian quotient of `F6` is empty, so `F6` is perfect.

Example H58E37

We construct an extension of A_5. This time the normal subgroup will be elementary abelian of order 2^5, with the action of A_5 being the natural permutation action.

```
> G := Alt(5);
> M := PermutationModule(G, GF(2));
> CohomologicalDimension(G, M, 2);
1
```

The dimension of the 2nd cohomology group is 1 over $\mathbf{F}_2$, so there are two possible extensions. We will construct them both.

```
> F := FPGroup(G);
> P := ExtensionProcess(G, M, F);
> E0 := Extension(P, [0]);
> E1 := Extension(P, [1]);
> AbelianQuotientInvariants(E0);
[ 2 ]
> AbelianQuotientInvariants(E1);
[]
```

The split extension, `E0`, is not perfect, but the non-split extension, `E1`, is a perfect group.

58.16 Representation Theory

A set of functions are provided for computing with the characters of a group. Full details of these functions may be found in Chapter 91. For convenience we include here two of the more useful character functions.

Also, functions are provided for computing with the modular representations of a group. Full details of these functions may be found in Chapter 89. For the reader's convenience we include here the functions which may be used to define a $R[G]$-module for a permutation group.

`CharacterTable(G: parameters)`

Construct the table of ordinary irreducible characters for the group G.

`Al` MonStgElt *Default :* "*Default*"

This parameter controls the algorithm used. The string `"DS"` forces use of the Dixon-Schneider algorithm. The string `"IR"` forces the use of Unger's induction/reduction algorithm [Ung06]. The `"Default"` algorithm is to use Dixon-Schneider for groups of order ≤ 5000 and Unger's algorithm for larger groups. This may change in future.

`DSSizeLimit` RngIntElt *Default :* 0

When the default algorithm is selected, a positive value n for `DSSizeLimit` means that before using Unger's algorithm, the full character space is split by some passes of Dixon-Schneider, restricted to using class matrices corresponding to conjugacy classes with size at most n.

`PermutationCharacter(G)`

Given a group G represented as a permutation group, construct the character of G afforded by the defining permutation representation of G.

`PermutationCharacter(G, H)`

Given a group G and some subgroup H of G, construct the ordinary character of G afforded by the permutation representation of G given by the action of G on the coset space of the subgroup H in G.

`GModule(G, S)`

Let G be a group defined on r generators and let S be a subalgebra of the matrix algebra $M_n(R)$, also defined by r non-singular matrices. It is assumed that the mapping from G to S defined by $\phi(G.i) \to S.i$, for $i = 1, \ldots, r$, is a group homomorphism. Let M be the natural module for the matrix algebra S. The function `GModule` gives M the structure of an $S[G]$-module, where the action of the i-th generator of G on M is given by the i-th generator of S.

`GModule(G, A, B)`

Given a finite group G, a normal subgroup A of G and a normal subgroup B of A such that the section A/B is elementary abelian of order p^n, create the $K[G]$-module M corresponding to the action of G on A/B, where K is the field $\mathbf{F}_p$. If B is trivial, it may be omitted. The function returns

(a) the module M; and

(b) the homomorphism $\phi : A/B \to M$.

`PermutationModule(G, H, R)`

Given a finite group G and a ring R, create the $R[G]$-module for G corresponding to the permutation action of G on the cosets of H.

`PermutationModule(G, R)`

Given a finite permutation group G and a ring R, create the natural permutation module for G over R.

Example H58E38

We refine an elementary abelian normal subgroup of a permutation group to a sequence of normal subgroups.

```
> G := PermutationGroup<24 |
>  [ 3, 4, 1, 2,23,24, 7, 8, 9,10,12,11,14,13,16,15,18,17,22,21,
>    20,19, 5, 6 ],
>  [ 7, 8,11,12,13,14,22,21,20,19,15,16,17,18, 6, 5, 4, 3, 1, 2,23,
>    24, 9,10 ] >;
> N := sub<G |
>  [ 24, 23, 6, 5, 4, 3, 10, 9, 8, 7, 14, 13, 12, 11, 18, 17, 16, 15, 22, 21,
>    20, 19, 2, 1 ],
```

```
>  [ 23, 24, 5, 6, 3, 4, 8, 7, 10, 9, 12, 11, 14, 13, 15, 16, 17, 18, 19, 20,
>    21, 22, 1, 2 ],
>  [ 2, 1, 4, 3, 6, 5, 7, 8, 9, 10, 11, 12, 13, 14, 17, 18, 15, 16, 21, 22, 19,
>    20, 24, 23 ]>;
> #N;
8
> IsNormal(G, N);
true
> IsElementaryAbelian(N);
true
> M, f := GModule(G, N);
> SM := Submodules(M);
> #SM;
4
> refined := [ x @@ f : x in SM ];
> forall{x : x in refined | IsNormal(G, x) };
true;
> [ #x : x in refined];
[ 1, 2, 4, 8 ]
```

The original elementary abelian normal subgroup of order 8 is the top of a chain of normal subgroups of length 3.

58.17 Identification

58.17.1 Identification as an Abstract Group

`NameSimple(G)`

Given a simple group G, determine the isomorphism type of G. The type is returned in the form of a triple of three integers f, d and q, where the interpretation of these integers is that given in the description of the function `CompositionFactors`.

58.17.2 Identification as a Permutation Group

The first functions described in this subsection detect whether or not a permutation group is alternating or symmetric in its natural representation. They are based on the algorithm 'Detect Alternating' outlined in [CB92].

`IsAlternating(G)`

Returns `true` if the permutation group G defined as acting on X is the alternating group $\mathrm{Alt}(X)$.

`IsSymmetric(G)`

Returns `true` if the permutation group G defined as acting on X is the symmetric group $\mathrm{Sym}(X)$.

`IsAltsym(G)`

Returns `true` if the permutation group G defined as acting on X contains the alternating group $\mathrm{Alt}(X)$.

`TwoTransitiveGroupIdentification(G)`

Given a 2-transitive group G, return a tuple giving the abstract isomorphism type of the group. This is an implementation of the method of Cameron and Cannon [CC91].

`RecogniseAlternatingOrSymmetric(G, n)`

Constructive recognition of the group G, which will succeed with probability $\geq 1-e^5$ if G is isomorphic to either the alternating or symmetric group of degree $n > 11$. The method is that of Beals *et al* [BLGN$^+$03], implemented by Colva Roney-Dougal.

The return values start with a flag indicating success or failure. If the algorithm was successful, then there are three more return values: a flag which is true when G is symmetric and false when alternating, and two programs. The first program takes an element x of an overgroup of G and produces a boolean to indicate whether $x \in G$ and a permutation representing x in the natural action of S_n (if such a permutation exists). The second taking a permutation to the corresponding element of G. The programs define mutually inverse group isomorphisms, implemented as MAGMA functions.

`IsEven(G)`

Given a permutation group G check if G is even, ie. contained in the alternating group.

Example H58E39

We give an example of `RecogniseAlternatingOrSymmetric` in use.

```
> SetSeed(1);
> a:= AlternatingGroup(13);
> h:= Stabiliser(a, {1,2});
> k:= CosetImage(a, h);
> Degree(k);
78
> worked, is_sym, bb_to_perm, perm_to_bb:=
> RecogniseAlternatingOrSymmetric(k, 13);
> worked;
true
> is_sym;
false
> x:= Sym(78)!(1, 35, 16, 28, 14, 26, 69, 5, 74)(2, 54,
> 67, 18, 51, 63, 6, 50, 77)(3, 33, 78, 12, 34, 29, 19, 15, 73)
> (4, 52, 61, 24, 49, 60, 68, 38, 64)(7, 20, 71, 17,
> 32, 11, 72, 8, 36)(9, 76, 47, 31, 56, 62, 13, 53, 59)
```

```
> (10, 70, 57, 23, 37, 22, 21, 27, 25)(30, 45, 46, 43, 42,
> 44, 40, 41, 75)(39, 55, 65)(48, 66, 58);
> x in k;
true;
> in_k, perm_image:= bb_to_perm(x);
> in_k;
true
> perm_image;
(1, 2, 3)(4, 7, 12, 6, 10, 11, 13, 9, 8)
> perm_to_bb(perm_image) eq x;
true
```

`RecogniseSymmetric(G, n: parameters)`

`maxtries`	RngIntElt	*Default :* $100n + 5000$
`Extension`	BoolElt	*Default :* `false`

The group G should be known to be isomorphic to the symmetric group S_n for some $n \geq 8$. The Bratus-Pak algorithm [BP00] (implemented by Derek Holt) is used to define an isomorphism between G and S_n. If successful, return `true`, homomorphism from G to S_n, homomorphism from S_n to G, the map from G to its word group and the map from the word group to G.

If the optional parameter `Extension` is set, then the group G should be known to be isomorphic either to S_n or to a perfect central extension $2.S_n$. In that case, the first two maps returned will be a homomorphism from G to S_n and a map from S_n to G that induces a homomorphism onto $G/Z(G)$. The sixth value returned will be `true`, if $G \cong 2.S_n$ and `false`, if $G \cong 2.A_n$.

If unsuccessful, `false` is returned. This will always occur if the input group is not isomorphic to S_n (or $2.S_n$ when `Extension` is set) with $n \geq 8$, and may occur occasionally even when G is isomorphic to S_n. The optional parameter `maxtries` (default $100n + 5000$) can be used to control the number of random elements chosen before giving up.

`SymmetricElementToWord (G, g)`

If g is an element of G which has been constructively recognised to be isomorphic to S_n (or $2.S_n$), then return `true` and element of word group for G which evaluates to g. Otherwise return `false`. This facilitates membership testing in G.

`RecogniseAlternating(G, n: parameters)`

`maxtries`	RngIntElt	*Default :* $100n + 5000$
`Extension`	BoolElt	*Default :* `false`

The group G should be known to be isomorphic to the alternating group A_n for some $n \geq 9$. The Bratus-Pak algorithm [BP00] (implemented by Derek Holt) is used to define an isomorphism between G and A_n. If successful, return `true`, homomorphism from G to A_n, homomorphism from A_n to G, the map from G to its word group and the map from the word group to G.

If the optional parameter `Extension` is set, then the group G should be known to be isomorphic either to A_n or to a perfect central extension $2.A_n$. In that case, the first two maps returned will be a homomorphism from G to A_n and a map from A_n to G that induces a homomorphism onto $G/Z(G)$. The sixth value returned will be `true`, if $G \cong 2.A_n$ and `false`, if $G \cong 2.A_n$.

If unsuccessful, `false` is returned. This will always occur if the input group is not isomorphic to A_n (or $2.A_n$ when `Extension` is set) with $n \geq 9$, and may occur occasionally even when G is isomorphic to A_n. The optional parameter `maxtries` (default $100n + 5000$) can be used to control the number of random elements chosen before giving up.

`AlternatingElementToWord (G, g)`

If g is an element of G which has been constructively recognised to be isomorphic to A_n (or $2.A_n$), then return `true` and element of word group for G which evaluates to g. Otherwise return `false`. This facilitates membership testing in G.

`GuessAltsymDegree(G: parameters)`

`maxtries`	RngIntElt	*Default :* 5000
`Extension`	BoolElt	*Default :* `false`

The group G should be believed to be isomorphic to S_n or A_n for some $n > 6$, or to $2.S_n$ or $2.A_n$ if the optional parameter `Extension` is set. This function attempts to determine n and whether G is symmetric or alternating. It does this by sampling orders of elements. It returns either `false`, if it is unable to make a decision after sampling `maxtries` elements (default 5000), or `true`, *type* and n, where *type* is "Symmetric" or "Alternating", and n is the degree. If G is not isomorphic to S_n or A_n (or $2.S_n$ or $2.A_n$ when `Extension` is set) for $n > 6$, then the output is meaningless - there is no guarantee that `false` will be returned. There is also a small probability of a wrong result or `false` being returned even when G is S_n or A_n with $n > 6$. This function was written by Derek Holt.

Example H58E40

For a group G which is believed to be isomorphic to S_n or A_n for some unknown value of $n > 6$, the function `GuessAltsymDegree` can be used to try to guess n, and then `RecogniseSymmetric` or `RecogniseAlternating` can be used to confirm the guess.

```
> SetSeed(1);
```

```
> G:= sub< GL(10,5) |
> PermutationMatrix(GF(5),Sym(10)![2,3,4,5,6,7,8,9,1,10]),
> PermutationMatrix(GF(5),Sym(10)![1,3,4,5,6,7,8,9,10,2]) >;
> GuessAltsymDegree(G);
true Alternating 10
> flag, m1, m2, m3, m4  := RecogniseAlternating(G,10);
> flag;
true
> x:=Random(G); Order(x);
8
> m1(x);
(1, 2, 4, 9, 10, 8, 6, 3)(5, 7)
> m2(m1(x)) eq x;
true
> m4(m3(x)) eq x;
true
> flag, w := AlternatingElementToWord(G,x);
> flag;
true
> m4(w) eq x;
true
> y := Random(Generic(G));
> flag, w := AlternatingElementToWord(G,y);
> flag;
false
> flag, m1, m2, m3, m4 := RecogniseAlternating(G,11);
> flag;
false
> flag, m1, m2, m3, m4 := RecogniseSymmetric(G,10);
> flag;
false
```

The nature of the `GuessAltsymDegree` function is that it assumes that its input is either an alternating or symmetric group and then tries to guess which one and the degree. As such, it is almost always correct when the input is an alternating or symmetric group, but will often return a bad guess when the input group is not of this form, as in the following example.

```
> GuessAltsymDegree(Sym(50));
true Symmetric 50
> GuessAltsymDegree(Alt(73));
true Alternating 73
> GuessAltsymDegree(PSL(5,5));
true Alternating 82
```

58.18 Base and Strong Generating Set

The key concept for representing a permutation group is that of a *base and strong generating set* (BSGS). Given a BSGS for a group, its order may be deduced immediately. Brownie, Cannon and Sims (1991) showed that it is practical, in some cases at least, to construct a BSGS for short-base groups having degree up to ten million.

The great majority of functions for computing with permutation groups require a BSGS to be present. If one is not known, MAGMA will attempt to automatically compute one. For large degree groups, the computation of a BSGS may be expensive and in such cases the user may achieve better performance through directly invoking a function which creates a BSGS. For example, if the group order is known in advance, it may be supplied to MAGMA and then a random method for computing a BSGS is applied which will use the group order as a termination condition.

In the first part of this section we present the elementary functions that use a BSGS, while towards the end we describe firstly, functions which allow the user to select and control the algorithm employed, and secondly, functions which provide access to the BSGS data structures. The material specific to BSGS should be omitted on a first reading.

58.18.1 Construction of a Base and Strong Generating Set

Computing structural information for a permutation group G requires, in most cases, a representation of the set of elements of G. MAGMA represents this set by means of a *base and strong generating set*, or *BSGS*, for G. Suppose the group G acts on the set Ω. A *base* B for G is a sequence of distinct points from Ω with the property that the identity is the only element of G that fixes B pointwise. A base B of length n determines a sequence of subgroups $G^{(i)}$, $1 \leq i \leq n+1$, where $G^{(i)}$ is the stabilizer of the first $i-1$ points of B. (In particular, $G^{(1)} = G$ and $G^{(n+1)}$ is trivial.) Given a base B for G, a subset S of G is said to be a *strong generating set* for G (with respect to B) if $G^{(i)} = \langle S \cap G^{(i)} \rangle$, for $i = 1, \ldots, n$.

`BSGS(G)`

The general procedure for constructing a BSGS. This version uses the default algorithm choices.

`SimsSchreier(G: parameters)`

`SV` BOOLELT *Default :* `true`

Construct a base and strong generating set for the group G using the standard Schreier-Sims algorithm. If the parameter `SV` is set `true` (default) the transversals are stored in the form of Schreier vectors. If `SV` is set `false`, then the transversals are stored both as lists of permutations and as Schreier vectors. If the base attribute has been previously defined for G, a variant of the Sims-Schreier algorithm will be employed, in which permutation multiplications are replaced by base image calculations wherever possible.

`RandomSchreier(G: parameters)`

`Max`	RngIntElt	*Default :* 100
`Run`	RngIntElt	*Default :* 20

Construct a probable base and strong generating set for the group G. The strong generators are constructed from a set of randomly chosen elements of G. The expectation is that, if sufficient random elements are taken, then, upon termination, the algorithm will have produced a BSGS for G. If the attribute Order is defined for G, the random Schreier will continue until a BSGS defining a group of the indicated order is obtained. In such circumstances this method is the fastest method of constructing a base and strong generating set for G. This is particularly so for groups of large degree. If nothing is known about G, the random Schreier algorithm provides a cheap way of obtaining lower bounds on the group's order and, in the case of a permutation group, on its degree of transitivity. This parameter has two associated parameters, `Max` and `Run`, which take positive integer values. The parameter `Max` specifies the number of random elements to be used (default 100). If the value of `Run` is n_2, then the algorithm terminates after n_2 consecutive random elements are found to lie in the set defined by the current BSGS (default 20). The two limits are independent of one another. It should be emphasized that unpredictable results may arise if the programmer uses the base and strong generators produced by this algorithm when, in fact, it does not constitute a BSGS for G.

`ToddCoxeterSchreier(G: parameters)`

Construct a BSGS for G using the Todd-Coxeter Schreier algorithm.

`SolubleSchreier(G: parameters)`

`SolvableSchreier(G: parameters)`

`Depth`	RngIntElt	*Default : See below*

Construct a base and strong generating set for the soluble permutation group G using the algorithm of Sims [Sim90]. The algorithm proceeds by recursively constructing the terms of the derived series. If G is not soluble then the algorithm will not terminate. In order to avoid non-termination, a limit on the number of terms in the normal subgroup chain constructed must be prescribed. The user may set this limit as the value of the parameter `Depth`. The default value, $\lceil 1.6 \ log_2 \ Degree(G) \rceil$, is based on an upper limit (due to Dixon) on the length of the derived series of a soluble permutation group. This algorithm is often significantly faster than the general Schreier-Sims algorithm.

`Verify(G: parameters)`

`Levels`	RngIntElt	*Default :* 0
`OrbitLimit`	RngIntElt	*Default :* 4,000

Given a permutation group G for which a probable BSGS is stored, verify the correctness of the BSGS. If it is not complete, proceed to complete it. The two parameters

`Levels` and `OrbitLimit` define how many levels the Todd-Coxeter-Schreier-Sims verifies before switching to the Brownie-Cannon-Sims algorithm. If `Levels` is set to n non-zero then n levels are verified by the TCSS algorithm before switching. If `Levels` is zero, the switch-over point is determined by the value of the parameter `OrbitLimit`. All levels with basic orbit length at most `OrbitLimit` are verified using TCSS. When a level is encountered with orbit length greater than this, a decision based on expected amount of work to do for this level by each algorithm determines what strategy is used for this level. Once one level uses the BCS method, all levels from then on will use it.

Example H58E41

The Higman-Sims simple group represented on 100 letters is generated by two permutations. To create a base and strong generating set for it using the Todd-Coxeter-Schreier algorithm, we can use the `ToddCoxeterSchreier` procedure as follows:

```
> G := sub<Sym(100) |
>    (2,8,13,17,20,22,7)(3,9,14,18,21,6,12)(4,10,15,19,5,11,16)
>        (24,77,99,72,64,82,40)(25,92,49,88,28,65,90)(26,41,70,98,91,38,75)
>        (27,55,43,78,86,87,45)(29,69,59,79,76,35,67)(30,39,42,81,36,57,89)
>        (31,93,62,44,73,71,50)(32,53,85,60,51,96,83)(33,37,58,46,84,100,56)
>        (34,94,80,61,97,48,68)(47,95,66,74,52,54,63),
>    (1,35)(3,81)(4,92)(6,60)(7,59)(8,46)(9,70)(10,91)(11,18)(12,66)(13,55)
>        (14,85)(15,90)(17,53)(19,45)(20,68)(21,69)(23,84)(24,34)(25,31)(26,32)
>        (37,39)(38,42)(40,41)(43,44)(49,64)(50,63)(51,52)(54,95)(56,96)(57,100)
>        (58,97)(61,62)(65,82)(67,83)(71,98)(72,99)(74,77)(76,78)(87,89) >;
> ToddCoxeterSchreier(G);
> Order(G);
44352000
```

Example H58E42

The simple group of Rudvalis has a permutation representation of degree 4060. A generating set for the Rudvalis group, Ru, may be found in the standard MAGMA database `pergps`, where it is called `ru`. We use the random Schreier algorithm followed by the `Verify` procedure to produce a base and strong generating set. We increase the limits for `RandomSchreier` to increase the probability that a complete base and strong generating set is found. This is done as follows:

```
> load "ru";
> RandomSchreier(G : Max := 50, Run := 20);
> Order(G);
145926144000
> Verify(G);
> Order(G);
145926144000
> Base(G);
[ 1, 2, 3, 4 ]
> BasicOrbitLengths(G);
```

```
[ 4060, 2304, 780, 20 ]
```

58.18.2 Defining Values for Attributes

If the order of a permutation group is known in advance, the construction of a base and strong generating set can be greatly speeded up by taking advantage of this knowledge. The `AssertAttribute` constructor may be used to communicate this and other useful information to Magma.

```
AssertAttribute(G, "Order", n)
```

Define the order attribute for the permutation group G.

```
AssertAttribute(G, "Order", Q)
```

Define the (factored) order of the permutation group G to be Q.

```
#AssertAttribute(G, "BSGS", S)
```

Define the base and strong generating set structure S to be the BSGS for G.

Example H58E43

The ability to set the order provides a short cut when constructing a BSGS. If the order attribute is set and the random Schreier-Sims algorithm applied, it will run until a BSGS for a group of the designated order has been constructed. We illustrate this in the case of the wreath product, with product action, of Sym(42) with Alt(8).

```
> G := WreathProduct(Sym(42), Alt(8));
> AssertAttribute(G, "Order", Factorial(42)^8 * (Factorial(8) div 2));
> RandomSchreier(G);
> Order(G);
306137301672361016520312745630712212453532972657840432\
7
542803407318869103099925605266692478702295013089037189\
1
052508966519418763874734093840686118134015088994465475\
2
702520725584513027443455822163386921007988258197506974\
2
167205546625091429100257027549968576864624041105576609\
8
237073911069065187521567666323553412613827078144015568\
3
990651535360000000000000000000000000000000000000000000\
0
000000000000000000000000000
```

58.18.3 Accessing the Base and Strong Generating Set

`Base(G)`

A base for the permutation group G. The base is returned as a sequence of points of its natural G-set. If a base is not known, one will be constructed.

`BasePoint(G, i)`

The i-th base point for the permutation group G. A base and strong generating set must be known for G.

`BasicOrbit(G, i)`

The basic orbit at level i as defined by the current base for the permutation group G. This function assumes that a BSGS is known for G.

`BasicOrbits(G)`

The basic orbits as defined by the current base for the permutation group G. This function assumes that a BSGS is known for G. The orbits are returned as a sequence of indexed sets.

`BasicOrbitLength(G, i)`

The length of the basic orbit at level i as defined by the current base for the permutation group G. This function assumes that a BSGS is known for G.

`BasicOrbitLengths(G)`

The lengths of the basic orbits as defined by the current base for the permutation group G. This function assumes that a BSGS is known for G. The lengths are returned as a sequence of integers.

`BasicStabilizer(G, i)`

`BasicStabiliser(G, i)`

Given a permutation group G for which a base and strong generating set are known, and an integer i, where $1 \leq i \leq k$ with k the length of the base, return the subgroup of G which fixes the first $i-1$ points of the base.

`BasicStabilizerChain(G)`

`BasicStabiliserChain(G)`

Given a permutation group G, return the stabilizer chain defined by the base as a sequence of subgroups of G. If a BSGS is not already known for G, it will be created.

`IsMemberBasicOrbit(G, i, a)`

Returns `true` if the point a of Ω lies in the basic orbit at level i. This function assumes that a BSGS is known for G.

`NumberOfStrongGenerators(G)`

`Nsgens(G)`

The number of elements in the current strong generating set for G.

`NumberOfStrongGenerators(G, i)`

`Nsgens(G, i)`

The number of elements in the strong generating set for the i-th term of the stabilizer chain for G.

`SchreierVectors(G)`

The Schreier vectors corresponding to the current BSGS for the permutation group G. The vectors are returned as a sequence of integer sequences.

`SchreierVector(G, i)`

The Schreier vector corresponding the i-th term of the stabilizer chain defined by the current BSGS for the permutation group G. The vector is returned as a sequence of integers.

`StrongGenerators(G)`

A set of strong generators for the permutation group G. If they are not currently available, they will be computed.

`StrongGenerators(G, i)`

A set of strong generators for the i-th term in the stabilizer chain for the permutation group G. A BSGS must be known for G.

58.18.4 Working with a Base and Strong Generating Set

`BaseImage(x)`

Given a permutation x belonging to the group G, for which a base and strong generating set is known, form the base image of x.

`Permutation(G, Q)`

Given a permutation group G acting on the set Ω, for which a base and strong generating set are known, and a sequence Q of distinct points of Ω defining an element x of G, return x as a permutation.

`SVPermutation(G, i, a)`

The permutation of G defined by the Schreier vector at level i, which takes the point a of Ω to the base point at level i. This function assumes that a BSGS is known for G.

`SVWord(G, i, a)`

An element in the word group of G defined by the Schreier vector at level i, which takes the point a of Ω to the base point at level i. This function assumes that a BSGS is known for G.

`Strip(H, x)`

Given an element x of a permutation group G, and given a group H for which a base and strong generating set is known, returns:

(a) the value of `x in H`

(b) The residual permutation y resulting from the stripping of x with respect to the BSGS for H; and

(c) The first level i such that y is not contained in $H^{(i)}$.

`WordStrip(H, x)`

Given an element x of a permutation group G, and given a group H for which a base and strong generating set is known, returns:

(a) the value of `x in H`

(b) the residual word w (an element in the word group of G) resulting from the stripping of x with respect to the BSGS for H,

(c) The first level i such that y is not contained in $H^{(i)}$.

`BaseImageWordStrip(H, x)`

Given an element x of a permutation group G, and given a group H for which a base and strong generating set is known, returns:

(a) Whether the base image strip succeeded at all levels. Note that a true value here does *not*, on its own, imply $x \in H$.

(b) the residual word w (an element in the word group of G) resulting from the stripping of x with respect to the BSGS for H,

(c) The first level i such that the strip could not continue.

`WordInStrongGenerators(H, x)`

Given an element x of a permutation group H for which a base and strong generating set is known, returns a word in the strong generators of H which represents x. This function uses base images to determine the word for x, so giving $x \notin H$ will have unpredictable results. This function returns the inverse of the second return value of `BaseImageWordStrip`, when the latter is successful.

58.18.5 Modifying a Base and Strong Generating Set

`ChangeBase(~G, Q)`

Given a group H with a base and strong generating set, change the base of G, so that the points in the sequence Q form an initial segment of the new base.

`AddNormalizingGenerator(~H, x)`

Given a group H with a base B and strong generating set X, and an element x that normalizes H belonging to a group that contains H, extend the existing BSGS for H so that they form a BSGS for the subgroup $< H, x >$.

`ReduceGenerators(~G)`

Given a group G with a base and strong generating set, remove redundant strong generators.

58.19 Permutation Representations of Linear Groups

Each of the functions in this family returns two values:

(a) A permutation group G corresponding to the action of a designated matrix group M on a vector space V; and

(b) An indexed set of affine or projective points on which M acts, such that the indexing gives the correspondence between this set and the G-set of M.

Furthermore, most of the function in this family are parameterized by two objects: the *degree* and the *coefficient field* of the matrix group. These can be supplied in one of the following three forms:

(i) Integers n and q corresponding to the degree and the field $\mathbf{F}_q$ of M ($\mathbf{F}_{q^2}$ in the case of the unitary groups).

(ii) An integer n and a finite field K corresponding to the degree and the coefficient field of M.

(iii) A vector space $V = K^n$ on which M naturally acts.

The Suzuki group, however, is only parametrised by the field, as the degree is always four. As such, it can be described by the integer q, the field $K = \mathbf{F}_q$, or the vector space K^4.

`AffineGeneralLinearGroup(`*arguments*`)`

`AGL(`*arguments*`)`

Construct the affine general linear group $G = \mathrm{AGL}(n, q)$, i.e., the group corresponding to the action of $\mathrm{GL}(n, q)$ on the affine points of the n-dimensional vector space V over $K = \mathbf{F}_q$. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the affine points and the G-set of G.

`AffineSpecialLinearGroup(`*arguments*`)`

`ASL(`*arguments*`)`

Construct the affine special linear group $G = \mathrm{ASL}(n, q)$, i.e., the group corresponding to the action of $\mathrm{SL}(n, q)$ on the affine points of the n-dimensional vector space V over $K = \mathbf{F}_q$. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the affine points and the G-set of G.

`AffineGammaLinearGroup(`*arguments*`)`

`AGammaL(`*arguments*`)`

Construct the affine gamma linear group $G = \mathrm{A\Gamma L}(n, q)$, i.e., the group corresponding to the action of $\Gamma\mathrm{L}(n, q)$ (the automorphism group of $\mathrm{GL}(n, q)$) on the affine points of the n-dimensional vector space V over $K = \mathbf{F}_q$. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`AffineSigmaLinearGroup(`*arguments*`)`

`ASigmaL(`*arguments*`)`

Construct the affine sigma linear group $G = \mathrm{A\Sigma L}(n, q)$, i.e., the group corresponding to the action of $\Sigma\mathrm{L}(n, q)$ (the automorphism group of $\mathrm{SL}(n, q)$) on the affine points of the n-dimensional vector space V over $K = \mathbf{F}_q$. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveGeneralLinearGroup(`*arguments*`)`

`PGL(`*arguments*`)`

Construct the projective general linear group $G = \mathrm{PGL}(n, q)$, i.e., the group corresponding to the action of $\mathrm{GL}(n, q)$ on the projective points of the n-dimensional vector space V over $K = \mathbf{F}_q$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSpecialLinearGroup(`*arguments*`)`

`PSL(`*arguments*`)`

Construct the projective special linear group $G = \mathrm{PSL}(n, q)$, i.e., the group corresponding to the action of $\mathrm{SL}(n, q)$ on the projective points of the n-dimensional vector space V over $K = \mathbf{F}_q$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveGammaLinearGroup(`*arguments*`)`

`PGammaL(`*arguments*`)`

Construct an automorphism group $G = \mathrm{P\Gamma L}(n, q)$ of the projective general linear group $B = \mathrm{PGL}(n, q)$, by adding the field automorphisms of $\mathbf{F}_q$ to B. The permutation action corresponds to the natural action on 1-dimensional subspaces of the n-dimensional vector space V over the field $K = \mathbf{F}_q$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveSigmaLinearGroup(`*arguments*`)`

`PSigmaL(`*arguments*`)`

Construct an automorphism group $G = \mathrm{P\Sigma L}(n, q)$ of the projective special linear group $B = \mathrm{PSL}(n, q)$, by adding the field automorphisms of $\mathbf{F}_q$ to B. The permutation action corresponds to the natural action on 1-dimensional subspaces of the n-dimensional vector space V over the field $K = \mathbf{F}_q$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveGeneralUnitaryGroup(`*arguments*`)`

`PGU(`*arguments*`)`

Construct the projective general unitary group $G = \mathrm{PGU}(n, q)$ corresponding to the n-dimensional vector space V over the field $K = \mathbf{F}_{q^2}$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSpecialUnitaryGroup(`*arguments*`)`

`PSU(`*arguments*`)`

Construct the projective special unitary group $G = \mathrm{PSU}(n, q)$ corresponding to the n-dimensional vector space V over the field $K = \mathbf{F}_{q^2}$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of V, giving the correspondence between these vectors and the G-set of G.

`ProjectiveGammaUnitaryGroup(`*arguments*`)`

`PGammaU(`*arguments*`)`

Construct an automorphism group $G = \mathrm{P\Gamma U}(n, q)$ of the projective general unitary group $B = \mathrm{PGU}(n, q)$, by adding the field automorphisms of $\mathbf{F}_{q^2}$ to B. The permutation action corresponds to the natural action on 1-dimensional subspaces of the n-dimensional vector space V over the field $K = \mathbf{F}_{q^2}$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveSigmaUnitaryGroup(`*arguments*`)`

`PSigmaU(`*arguments*`)`

Construct the automorphism group $G = \mathrm{P\Sigma U}(n, q)$ of the projective special unitary group $B = \mathrm{PSU}(n, q)$, by adding the field automorphisms of $\mathbf{F}_{q^2}$ to B. The permutation action corresponds to the natural action on 1-dimensional subspaces of the n-dimensional vector space V over the field $K = \mathbf{F}_{q^2}$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveSymplecticGroup(`*arguments*`)`

`PSp(`*arguments*`)`

Construct the projective symplectic group $G = \mathrm{PSp}(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 4. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSigmaSymplecticGroup(`*arguments*`)`

`PSigmaSp(`*arguments*`)`

Construct the group $G = \mathrm{P\Sigma Sp}(n, q)$ of the projective symplectic group $\mathrm{PSp}(n, q)$ extended by field automorphisms of $K = \mathbf{F}_q$, where V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 4. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveGeneralOrthogonalGroup(`*arguments*`)`

`PGO(`*arguments*`)`

Construct the projective general orthogonal group $G = \mathrm{PGO}(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an odd integer greater than or equal to 3. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveGeneralOrthogonalGroupPlus(`*arguments*`)`

`PGOPlus(`*arguments*`)`

Construct the projective general orthogonal group $G = \mathrm{PGO}^+(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveGeneralOrthogonalGroupMinus(`*arguments*`)`

`PGOMinus(`*arguments*`)`

Construct the projective general orthogonal group $G = \mathrm{PGO}^-(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSpecialOrthogonalGroup(`*arguments*`)`

`PSO(`*arguments*`)`

Construct the projective special orthogonal group $G = \mathrm{PSO}(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an odd integer greater than or equal to 3. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSpecialOrthogonalGroupPlus(`*arguments*`)`

`PSOPlus(`*arguments*`)`

Construct the projective special orthogonal group $G = \mathrm{PSO}^+(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSpecialOrthogonalGroupMinus(`*arguments*`)`

`PSOMinus(`*arguments*`)`

Construct the projective general orthogonal group $G = \mathrm{PSO}^-(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveOmega(`*arguments*`)`

`POmega(`*arguments*`)`

Construct the projective orthogonal group $G = \mathrm{P\Omega}(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an odd integer greater than or equal to 3. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveOmegaPlus(`*arguments*`)`

`POmegaPlus(`*arguments*`)`

Construct the projective orthogonal group $G = \mathrm{P\Omega}(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveOmegaMinus(`*arguments*`)`

`POmegaMinus(`*arguments*`)`

Construct the projective orthogonal group $G = \mathrm{P}\Omega(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSuzukiGroup(`*arguments*`)`

`PSz(`*arguments*`)`

Construct the permutation representation $G = \mathrm{PSz}(q)$ of the Suzuki simple group $\mathrm{Sz}(q)$, given by its action on projective points, where q is of the form 2^{2n+1}. If K is given, its cardinality is q. If V is given, it must be 4-dimensional, and over K. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`AffineGroup(M)`

Given a matrix group of degree d over a finite field F, construct the semidirect product $V : M$, where $V = F^d$ is the natural M-module. The result, G, is a standard permutation group of degree $|V| = |F|^d$, where the second return value gives the correspondence between the elements of V and the standard G-set.

58.20 Permutation Group Databases

Magma includes databases that contain all transitive permutation groups of degree up to 32 and all primitive permutation groups of degree up to 4095. Descriptions of these databases may be found in Chapter 66.

58.21 Ordered Partition Stacks

Ordered partition stacks have been implemented with their own type and Magma intrinsics. They implement the data structure described very briefly in section 2 of Jeff Leon's 1997 paper [Leo97]. They can be used as an aid to implementing various backtrack searches in the Magma language.

The domain set of the partitions is always $\{1..n\}$, where n is called the degree of the stack. The basic "push" operation for these stacks involves refining an ordered partition, and the precise definition of refinement used in the Magma implementation is Definition 2 of [Leo97]. This differs from the definition in Chapter 9 of [Ser03], for instance.

The word "ordered" refers to the cells of the partition being in a fixed order. The order of points in a cell is not significant, and may vary as the data structure is manipulated.

58.21.1 Construction of Ordered Partition Stacks

`OrderedPartitionStack(n)`

Create a data structure representing a complete ordered partition stack of degree n. Initially the stack has one partition on it, which is the partition having a single block.

`OrderedPartitionStackZero(n, h)`

Create a data structure representing a zero-based ordered partition stack of degree n with height limited to h. Initially the stack has one partition on it, which is the partition having a single block, and has height 0.

58.21.2 Properties of Ordered Partition Stacks

`Degree(P)`

The degree of the ordered partition stack P.

`Height(P)`

The height of the ordered partition stack P. For a complete stack, this equals the number of cells of the finest partition on the stack.

`NumberOfCells(P, h)`

The number of cells in the partition on stack P at height h. If h is omitted it is taken to be the height of P, so giving the number of cells in the finest partition on the stack P.

`CellNumber(P, h, x)`

The number of the cell of the partition at height h in P that contains the element x. If h is omitted it is taken to be the height of P.

`CellSize(P, h, i)`

The size of cell i of the partition at height h in P. If h is omitted it is taken to be the height of P.

`Cell(P, h, i)`

The contents of cell i of the partition at height h in P as a sequence of integers. If h is omitted it is taken to be the height of P. Note that the order of the points in the returned sequence may vary, as the order of points in a cell of an ordered partition is not fixed.

`Random(P, i)`

A random element of cell i of the finest partition on P.

`Representative(P, i)`

`Rep(P, i)`

An element of cell i of the finest partition on P.

`ParentCell(P, i)`

The number of the cell that was split to first create cell number i.

58.21.3 Operations on Ordered Partition Stacks

Here are listed the basic operations provided for pushing a finer partition onto an ordered partition stack, called splitting, and for popping ordered partitions off the stack.

If the top partition on the stack has k cells, and one of these cells is split to form a finer partition, then the new cell will have number $k + 1$, and the residue of the split cell will have the same number as the cell that was split. This agrees with the definition of refinement given in [Leo97], Definition 2, but disagrees with [Ser03], Chapter 9.2, and [McK81].

`SplitCell(P, i, x)`

`SplitCell(P, i, Q)`

Attempt to refine the top partition on stack P by splitting cell i. The new cell created will be $\{x\}$ if x is in cell i, or will be the intersection of Q with cell i (in the second form), if this is not empty and not all of cell i. The new partition, if any, is pushed onto the stack. The return value is true when P is changed, false otherwise. This implements the operation in Definition 6 of [Leo97].

`SplitAllByValues(P, V)`

Refine the top partition on stack P by splitting all possible cells using the values in V. This implements the operation given in Definition 15 of [Leo97]. Cells are split in increasing order of cell number, and the resulting new cells are in the curious order given in the cited definition.

The first return value is true when P is changed, false otherwise. The second is a hash value based on which cells are split, the values from V used in the split, and the sizes of the resulting cells. It is suitable for use as an indicator function, as defined in 2-16 of [McK81].

`SplitCellsByValues(P, C, V)`

`SplitCellsByValues(P, i, V)`

Refine the top partition on stack P by splitting all cells given in C, or cell i, using the values in V. Splitting and return values are as for `SplitAllByValues`, with an important difference: cells will be split in the order given in C, and, if some cell in C does not split, the operation will be terminated there, and false returned.

`Pop(P)`

`Pop(P, h)`

Reduce the height of the partition stack P to height h, or by one if h is not given. The method used is the "retract" algorithm of [Leo97], Fig. 7.

`Advance(X, L, P, h)`

This implements the "advance" algorithm of [Leo97], Fig. 7: X is a zero-based stack of degree d say, L is a sequence of length n taking values in $\{1..d\}$, representing an unordered partition of $\{1..n\}$ into d blocks, P is a complete stack of degree n, and h is a positive integer which is at most the height of P. This is a fundamental operation in Leon's unordered partition stabilizer algorithm.

Example H58E44

We set up an ordered partition stack of degree 12, and try out a few basic operations on it. The printing of a stack shows the top partition of the stack.

```
> P := OrderedPartitionStack(12);
> P;
Partn stack, degree 12, height 1
[ 1 2 3 4 5 6 7 8 9 10 11 12]
> SplitCell(P, 1, 4);
true
> P;
Partn stack, degree 12, height 2
```

```
[ 1 2 3 12 5 6 7 8 9 10 11 | 4]
```

Note that the order of the points in cell 1 is not significant. Now we will split on the values in a vector V.

```
> V := [i mod 5 + 1: i in [0..11]];
> V;
[ 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2 ]
> SplitAllByValues(P, V);
true 119375796
> P;
Partn stack, degree 12, height 6
[ 1 6 11 | 4 | 10 5 | 9 | 8 3 | 12 2 7]
```

Only cell 1 has been split. The points corresponding to the minimum value in V remain in cell 1. The new cells are cells 2 to 6. They correspond to the higher values in V, in descending order. Now pop the stack back to height 4 and try the effect of a different split by values.

```
> Pop(P, 4);
> P;
Partn stack, degree 12, height 4
[ 1 6 11 12 2 7 8 3 | 4 | 10 5 | 9]
> V := [i mod 4 + 1: i in [0..11]];
> V;
[ 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4 ]
> SplitAllByValues(P, V);
true 985543242
> P;
Partn stack, degree 12, height 8
[ 1 | 4 | 5 | 9 | 8 12 | 11 7 3 | 6 2 | 10]
> Pop(P, 3);
> P;
Partn stack, degree 12, height 3
[ 1 6 2 11 7 3 8 12 9 | 4 | 5 10]
```

58.22 Bibliography

[**Atk75**] M.D. Atkinson. An algorithm for finding the blocks of a permutation group. *Math. Comp.*, 29:911–913, 1975.

[**BC82**] G. Butler and J.J. Cannon. Computing with permutation and matrix groups I: Normal closure, commutator subgroups, series. *Math. Comp.*, 39:671–680, 1982.

[**Bea93**] G. Beals. *Algorithms for finite groups.* PhD thesis, University of Chicago, 1993.

[**BLGN$^+$03**] R. Beals, C. R. Leedham-Green, A. C. Niemeyer, C. E. Praeger, and A. Seress. A black-box algorithm for recognising finite symmetric and alternating groups, I. *Trans. Amer. Math. Soc.*, 2003. To appear.

[**BP00**] Sergey Bratus and Igor Pak. Fast constructive recognition of a black box group isomorphic to S_n or A_n using Goldbach's conjecture. *J. Symbolic Comp.*, 29:33–57, 2000.

[**But85**] Gregory Butler. Effective computation with group homomorphisms. *J. Symbolic Comp.*, 1:143–157, 1985.

[**But94**] Greg Butler. An inductive schema for computing conjugacy classes in permutation groups. *Mathematics of Computation*, 62(205):363–383, 1994.

[**CB92**] J.J. Cannon and W. Bosma. Structural computation in finite permutation groups. *CWI Quarterly*, 5(2):127–160, 1992.

[**CC91**] P.J. Cameron and J.J. Cannon. Recognizing doubly transitive groups. *J. Symb. Comp.*, 12(4/5):459–474, 1991.

[**CCH97**] J.J. Cannon, B. Cox, and D.F. Holt. Computing Sylow subgroups in permutation groups. *J. Symb. Comp.*, 24(3/4):303–316, 1997.

[**CCH01**] J.J. Cannon, B. Cox, and D.F. Holt. Computing the subgroups of a permutation group. *J. Symb. Comp.*, 31:149–161, 2001.

[**CFL89**] G. Cooperman, L. Finkelstein, and E.M. Luks. Reduction of group constructions to point stabilizers. In *Proc. of International Symposium on Symbolic and Algebraic Computation ISSAC '89*, pages 351–356. ACM, 1989.

[**CH97**] J.J. Cannon and D.F. Holt. Computing chief series, composition series and socles in large permutation groups. *J. Symb. Comp.*, 24(3/4):285–301, 1997.

[**CH03**] J.J. Cannon and D.F. Holt. Automorphism group computation and isomorphism testing in finite groups. *J. Symbolic Comp.*, 35(3):241–267, 2003.

[**CH04**] J.J. Cannon and D.F. Holt. Computing maximal subgroups of finite groups. *J. Symbolic Comp.*, 37(5):589–609, 2004.

[**CHSS03**] J.J. Cannon, D.F. Holt, M. Slattery, and A.K. Steel. Computing subgroups of low index in a finite group. 2003.

[**CLGM$^+$95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

[**CS**] J.J. Cannon and B. Souvignier. On the computation of normal subgroups in permutation groups. to appear, International Journal of Algebra and Computation.

[**CS97**] J.J. Cannon and B. Souvignier. On the computation of conjugacy classes in permutation groups. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, pages 392–399. Association for Computing Machinery, 1997. Maui, July 21–23, 1997.

[**Geb00**] Volker Gebhardt. Constructing a short defining set of relations for a finite group. *J. Algebra*, 233:526–542, 2000.

[**Hol84**] D.F. Holt. The calculation of the Schur multiplier of a permutation group. In *Computational group theory (Durham, 1982)*, pages 307–319. Academic Press, London, 1984.

[**Hol85a**] D.F. Holt. A computer program for the calculation of a covering group of a finite group. *J. Pure Appl. Algebra*, 35(3):287–295, 1985.

[**Hol85b**] D.F. Holt. The mechanical computation of first and second cohomology groups. *J. Symbolic Comp.*, 1(4):351–361, 1985.

[**Kan91**] William M. Kantor. Finding composition factors of permutation groups of degree $n \leq 10^6$. *J. Symbolic Comp.*, 12(4/5):517–526, 1991.

[**Leo80**] Jeffrey S. Leon. On an algorithm for finding a base and a strong generating set for a group given by generating permutations. *Math. Comp.*, 35(151):941–974, 1980.

[**Leo97**] Jeffrey S. Leon. Partitions, refinements, and permutation group computation. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of *Dimacs series in Discrete Mathematics and Computer Science*, pages 123–158, Providence R.I., 1997. Amer. Math. Soc.

[**LGPS91**] C.R. Leedham-Green, C.E. Praeger, and L.H. Soicher. Computing with group homomorphisms. *J. Symbolic Comp.*, 12(4/5):527–532, 1991.

[**LNPS06**] M. Law, A.C. Niemeyer, C.E. Praeger, and A. Seress. A reduction algorithm for for large-base primitive permutation groups. *LMS J. Comput. Math.*, 9:159173, 2006.

[**Luk93**] E.M. Luks. Permutation groups and polynomial-time computation. In *Groups and computation (New Brunswick, NJ, 1991)*, volume 11 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 139–175. Amer. Math. Soc., 1993.

[**McK81**] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[**MN89**] M. Mecky and J. Neubüser. Some remarks on the computation of conjugacy classes of soluble groups. *Bull. Austral, Math. Soc.*, 40(2):281–292, 1989.

[**Neu86**] P.M. Neumann. Some algorithms for computing with finite permutation groups. In C.M. Campbell E.F. Robertson, editor, *Groups - St. Andrews 1985*, number 121 in London Math. Soc. Lecture Notes Series, 1986.

[**Ric73**] J.S. Richardson. Group: a computer system for group-theoretic calculations. Master's thesis, Department of Pure Mathematics, University of Sydney, September 1973.

[**Sch90**] Bernd Schmalz. Verwendung von Untergruppenleitern zur Bestimmung von Doppelnebenklassen. *Bayreuther Mathematische Schriften*, 31:109–143, 1990.

[**Ser03**] Ákos Seress. *Permutation Group Algorithms*, volume 152 of *Cambridge Tracts in Mathematics*. Cambridge University Press, Cambridge, 2003.

[**Sim90**] Charles C. Sims. Computing the order of a solvable permutation group. *J. Symb. Comp.*, 9(5/6):699–705, 1990.

[**SS94**] M. Schönert and A. Seress. Finding blocks of imprimitivity in small-base groups in nearly linear time. In *Proc. 1994 ACM-SIGSAM Inter. Symp. on Symbolic and Algebraic Comp.*, pages 154–157, 1994.

[**Ung**] W.R. Unger. Computing chief series of a large permutation group. In preparation.

[**Ung06a**] W.R. Unger. Computing the character table of a finite group. *J. Symbolic Comp.*, 41(8):847–862, 2006.

[**Ung06b**] W.R. Unger. Computing the solvable radical of a permutation group. *J. Algebra*, 300(1):305–315, 2006.

59 MATRIX GROUPS OVER GENERAL RINGS

Chapter 59
MATRIX GROUPS OVER GENERAL RINGS

59.1 Introduction

59.1.1 Introduction to Matrix Groups

A matrix group G may be defined over any ring R for which MAGMA has a method for computing the inverse of a matrix. However, the availability of machinery for determining structural information is dependent upon the properties of the base ring R.

We distinguish several different cases.

(i) If the ring R is a finite field then the group must be finite. If the group has moderate degree and it is possible to find a low dimensional subspace of the natural vector space for G whose orbit under G has length bounded by a million or so, then it is possible to construct a stabilizer chain representation for the group similar to that used for permutation groups (the *BSGS representation* [But76]). In order to increase the chances of finding a short orbit the Murray-O'Brien [MO95] strategy for selecting base points is used. The availability of a BSGS representation allows the structure of the group to be investigated in detail.

(ii) If the coefficient ring R is a finite field but the degree and size of the group are such that it is not possible to construct a useful BSGS representation then the group may be investigated using techniques based on a theorem of Aschbacher that classifies the maximal subgroups of $\mathrm{GL}(n, q)$. This approach is under intensive development by Leedham-Green, O'Brien and others. Code implementing some parts is documented in Chapter 65.

(iii) If the ring R is the Euclidean Ring $\mathbf{Z}/m\mathbf{Z}$, then the group must be finite. If the group has moderate degree and it is possible to find a vector in the natural R-module for G whose orbit has length bounded by a million or so, then again it is possible to compute structural information using the BSGS representation.

(iv) If the ring R has infinite cardinality and satisfies certain properties, MAGMA can sometimes determine whether the group is finite or infinite. In particular, this can be done when R is the ring of integers, the field of rational numbers, or an algebraic number field (including cyclotomic and quadratic fields). If G is infinite, and has not been created as a Lie group, then MAGMA currently provides little beyond basic arithmetic on elements.

(v) If the ring R has infinite cardinality but the group G is finite and R is either a field or an Euclidean Domain then it may be possible to construct a *BSGS representation* as above and thereby undertake structural computation.

(vi) If an (infinite) matrix group can be created as a Lie group then machinery based on Lie Theory may be used to analyse the group. The facilities for Lie groups are described in Chapter 103.

Matrix groups over rings of infinite cardinality may be created regardless as to whether they are finite or not. If the coefficient ring R is either the ring of integers, the rational field, a quadratic field, a cyclotomic field, or a number field a matrix group may then be tested for finiteness by use of the function `IsFinite`. However, most functions that determine structural properties of a group apply only to finite groups.

59.1.2 The Support

Matrix groups may be defined over any ring for which MAGMA has a method for computing matrix inverses. However, the structure algorithms assume that the group is finite and is defined over either a field, an Euclidean Domain or the Euclidean Ring $\mathbf{Z}/m\mathbf{Z}$.

59.1.3 The Category of Matrix Groups

The family of matrix groups over a particular ring R forms a category where the objects are the matrix groups and the morphisms are group homomorphisms. The collection of all matrix groups forms a family of categories indexed by the category of rings. The MAGMA designation for this family of categories of matrix groups is `GrpMat`.

59.1.4 The Construction of a Matrix Group

A group of $n \times n$ matrices defined over the ring R is created as a subgroup of the general linear group $\mathrm{GL}(n, R)$. Thus the construction of a general matrix group is a two step process:

(i) The appropriate general linear group, $\mathrm{GL}(n, R)$, is constructed;

(ii) The required group G is then defined as a subgroup of $\mathrm{GL}(n, R)$.

For convenience, a constructor `MatrixGroup< ... >`, which combines these two steps, is provided.

59.2 Creation of a Matrix Group

59.2.1 Construction of the General Linear Group

`GeneralLinearGroup(n, R)`

`GL(n, R)`

Given an integer $n \geq 1$ and a ring R, create the generic matrix group, i.e. the general linear group $\mathrm{GL}(n, R)$. Initially, only a structure table is created for $\mathrm{GL}(n, R)$, so that, in particular, generators are not defined. This function is normally used to provide a context for the creation of elements and subgroups of $\mathrm{GL}(n, R)$. If structural computation is attempted with the group created by `GeneralLinearGroup(n, R)`, then generators will be created where possible. At present, this is only permitted in the cases in which R is a finite field.

Example H59E1

We define the general linear group $\mathrm{GL}(3, K)$, where K is the finite field $\mathbf{F}_4$.

```
> K<w> := FiniteField(4);
> GL34 := GeneralLinearGroup(3, K);
> GL34;
GL(3, GF(2, 2))
```

59.2.2 Construction of a Matrix Group Element

Throughout this subsection we shall assume that the matrix group G is defined over the ring R.

`elt< G | L >`

Given a matrix group G defined as a subgroup of $\mathrm{GL}(n, R)$, and the list L of expressions a_{ij} $(1 \leq i, j \leq n)$, defining elements of the ring R, construct the $n \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

Unless G is known to be the generic matrix group of degree n, the matrix will be tested for membership of G, and if g is not an element of G, the function will fail. If g does lie in G, g will have G as its parent. Since the membership test may involve constructing a base and strong generating set for G, this constructor may occasionally be very costly. Hence a matrix g should be defined as an element of a subgroup of the generic group only when membership of G is required by subsequent operations involving g.

`G ! Q`

Given the sequence Q of expressions a_{ij} $(1 \leq i, j \leq n)$, defining elements of the ring R, construct the $n \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

This matrix will have G as its parent structure. As in the case of the `elt`-constructor, the operation will fail if g is not an element of G, and the same observations concerning the cost of membership testing apply.

`ElementToSequence(g)`

`Eltseq(g)`

Given an $n \times n$ matrix $g = (a_{ij}), 1 \leq i, j \leq n$, where a_{ij} is an element of the ring R, construct the sequence

$$[a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots a_{n1}, \ldots, a_{nn}]$$

of n^2 elements of the ring R.

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity matrix in the matrix group G.

Example H59E2

The different constructions are illustrated by the following code, which assigns to each of the variables x and y an element of $\mathrm{GL}(3, 4)$.

```
> K<w> := FiniteField(4);
> GL34 := GeneralLinearGroup(3, K);
> x := elt<GL34 | w,0,1, 0,1,0, 1,0,1 >;
> x;
[w 0 1]
[0 1 0]
[1 0 1]
> y := GL34 ! [w,0,1, 0,1,0, 1,0,1];
> y;
[w 0 1]
[0 1 0]
[1 0 1]
> GL34 ! 1;
[1 0 0]
[0 1 0]
[0 0 1]
```

59.2.3 Construction of a General Matrix Group

`MatrixGroup< n, R | L >`

Construct the matrix group G of degree n over the ring R generated by the matrices defined by the list L. A term of the list L must be an object of one of the following types:

(a) A sequence of n^2 elements of R defining a matrix of $\mathrm{GL}(n, R)$;

(b) A set or sequence of sequences of type (a);

(c) An element of $\mathrm{GL}(n, R)$;

(d) A set or sequence of elements of $\mathrm{GL}(n, R)$;

(e) A subgroup of $\mathrm{GL}(n, R)$;

(f) A set or sequence of subgroups of $\mathrm{GL}(n, R)$.

Each element or group specified by the list must belong to the *same* generic matrix group. The group G will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of G consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list. Repetitions of an element and occurrences of the identity element are removed.

The `MatrixGroup` constructor is shorthand for the two statements:

```
GL := GeneralLinearGroup(n, R);
G := sub< GL | L >;
```

where `sub< ... >` is the subgroup constructor described in the next subsection.

Example H59E3

We use the `MatrixGroup` constructor to define a small subgroup of $\mathrm{GL}(3, 4)$.

```
> K<w> := FiniteField(4);
> H := MatrixGroup< 3, K | [1,w,0, 0,1,0, 1,w^2,1], [w,0,0, 0,1,0, 0,0,w] >;
> H;
MatrixGroup(3, GF(2, 2))
Generators:
[  1   w   0]
[  0   1   0]
[  1 w^2   1]

[w 0 0]
[0 1 0]
[0 0 w]
> Order(H);
96
```

Example H59E4

We present a function which will construct the Sylow p-subgroup of $\mathrm{GL}(n, K)$, where K is a finite field of characteristic p.

```
> GLSyl := function(n, K)
>    R := MatrixRing(K, n);
>    e := func< i, j | MatrixUnit(R, i, j) >;
>    return MatrixGroup< n, K | { R!1 + a*e(i,j) : a in K, j in [i+1],
>          i in [1 .. n - 1] | a ne 0 } >;
> end function;
> T := GLSyl(3, GF(8));
> FactoredOrder(T);
[ <2, 9> ]
> FactoredOrder(GL(3, GF(8)));
[ <2, 9>, <3, 2>, <7, 3>, <73, 1> ]
```

59.2.4 Changing Rings

`ChangeRing(G, S)`

Given a matrix group G with base ring R, construct a new matrix group H with base ring S derived from G by coercing entries of the generators of G from R into S.

`ChangeRing(G, S, f)`

Given a matrix group G with base ring R, construct a new matrix group H with base ring S derived from G by applying f to the entries of the generators of G.

`RestrictField(G, S)`

Given a matrix group G with base ring K, a finite field, and S a subfield of K, construct the matrix group H with base ring S obtained by restricting the scalars of the components of elements of G into S, together with the restriction map from G onto H.

`ExtendField(G, L)`

Given a matrix group G with base ring K, a finite field, and L an extension of K, construct the matrix group H with base ring L obtained by lifting the components of elements of G into L, together with the inclusion homomorphism from G into H.

59.2.5 Coercion between Matrix Structures

A square non-singular matrix may be defined as an element of any of the following structures:

- A subring of the complete matrix ring $M_n(R)$;
- A subgroup of the general linear group $\mathrm{GL}(n, R)$;
- A submodule of the matrix module $M^{(m \times n)}(R)$.

The coercion operator may be used to transfer matrices between any two of these three structures.

`R ! g`

Transfer the matrix g from a group into a matrix ring R.

`G ! r`

Transfer the matrix r from a ring into a matrix group G.

`M ! g`

Transfer the matrix g from a group into a matrix module M.

`G ! m`

Transfer the matrix m from a module into a matrix group G.

59.2.6 Accessing Associated Structures

The functions in this group provide access to basic information stored for a matrix group G.

`G . i`

The i-th defining generator for the matrix group G. A negative subscript indicates that the inverse of the generator is to be created. $G.0$ is `Identity(G)`.

`Degree(G)`

The degree of the matrix group G.

`Generators(G)`

A set containing the defining generators for the matrix group G.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for the matrix group G.

`CoefficientRing(G)`

`BaseRing(G)`

The coefficient ring for the matrix group G.

`RSpace(G)`

Given a matrix group G of degree n defined over a ring R, return the space $R^{(n)}$, where the action is multiplication by elements of R, i.e. scalar action.

`VectorSpace(G)`

Given a matrix group G of degree n defined over a field K, return the space $K^{(n)}$, where the action is multiplication by elements of K, i.e. scalar action.

`GModule(G)`

The natural $R[G]$-module for the matrix group G.

`Generic(G)`

The generic group containing the matrix group G, i.e. the general linear group in which G is naturally embedded.

`Parent(G)`

The power structure for the group G (the set consisting of all matrix groups).

59.3 Homomorphisms

Homomorphisms are an important part of group theory, and MAGMA supports homomorphisms between groups. Many useful homomorphisms are returned by constructors and intrinsic functions. Examples of these are the `quo` constructor, the `sub` constructor and intrinsic functions such as `OrbitAction` and `FPGroup`, which are described in more detail elsewhere in this chapter. In this section we describe how the user may create their own homomorphisms with domain a matrix group.

`hom< G - >`

Given the matrix group G, construct the homomorphism $f : G \to H$ given by the generator images in L. H must be a group. The clause L may be any one of the following types:

(a) A list of elements of H, giving images of the generators of G;

(b) A list of pairs, where the first in the pair is an element of G and the second its image in H, where pairs may be given in either of the (equivalent) forms `<g,h>` or `g -> h`;

(c) A sequence of elements of H, as in (a);

(d) A set or sequence of pairs, as in (b);

Each image element specified by the list must belong to the *same* group H. In the cases where pairs are given the given elements of G must generate G.

`Domain(f)`

The domain of the homomorphism f.

`Codomain(f)`

The codomain of the homomorphism f.

`Image(f)`

The image or range of the homomorphism f. This will be a subgroup of the codomain of f. The algorithm computes the image and kernel simultaneously (see [LGPS91]).

`Kernel(f)`

The kernel of the homomorphism f. This will be a normal subgroup of the domain of f. The algorithm computes the image and kernel simultaneously (see [LGPS91]).

`IsHomomorphism(G, H, Q)`

Return the value `true` if the sequence Q defines a homomorphism from the group G to the group H. The sequence Q must have length `Ngens(G)` and must contain elements of H. The i-th element of Q is interpreted as the image of the i-th generator of G and the function decides if these images extend to a homomorphism. If so, the homomorphism is also returned.

Example H59E5

We construct the usual degree 2 matrix representation of the dihedral group of order 20, and a homomorphism from it to the symmetric group of degree 5.

```
> K<z> := CyclotomicField(20);
> zz := RootOfUnity(10, K);
> i := RootOfUnity(4, K);
> cos := (zz+ComplexConjugate(zz))/2;
> sin := (zz-ComplexConjugate(zz))/(2*i);
> gl := GeneralLinearGroup(2, K);
> M := sub< gl | [cos, sin, -sin, cos], [-1,0,0,1]>;
> #M;
20
> S := SymmetricGroup(5);
> f := hom<M->S |[S|(1,2,3,4,5), (1,5)(2,4)]>;
> Codomain(f);
Symmetric group S acting on a set of cardinality 5
Order = 120 = 2^3 * 3 * 5
> Image(f);
Permutation group acting on a set of cardinality 5
Order = 10 = 2 * 5
  (1, 2, 3, 4, 5)
  (1, 5)(2, 4)
> Kernel(f);
MatrixGroup(2, K) of order 2
Generators:
  [-1  0]
```

```
[ 0 -1]
```

59.3.1 Construction of Extensions

`DirectProduct(G, H)`

Given two matrix groups G and H of degrees m and n respectively, construct the direct product of G and H as a matrix group of degree $m + n$.

`DirectProduct(Q)`

Given a sequence Q of n matrix groups, construct the direct product $Q[1] \times Q[2] \times \ldots \times Q[n]$ as a matrix group of degree equal to the sum of the degrees of the groups $Q[i]$, $(i = 1, \ldots, n)$.

`SemiLinearGroup(G, S)`

Given a matrix group G over the finite field K and a subfield S of K, construct the semilinear extension of G over the subfield S.

`TensorWreathProduct(G, H)`

Given a matrix group G and a permutation group H, construct action of the wreath product on the tensor power of G by H, which is the (image of) the wreath product in its action on the tensor power (of the space that G acts on). The degree of the new group is d^k where d is the degree of G and k is the degree of H.

`WreathProduct(G, H)`

Given a matrix group G and a permutation group H, construct the wreath product $G \wr H$ of G and H.

Example H59E6

We define G to be $\mathrm{SU}(3, 4)$ and H to be the symmetric group of order 6. We then proceed to form the direct product of G with itself and the tensor and wreath products of G and H.

```
> K<w> := FiniteField(4);
> G := SpecialUnitaryGroup(3, K);
> D := DirectProduct(G, G);
> D;
MatrixGroup(6, GF(2, 2))
Generators:
[  1   w   w   0   0   0]
[  0   1 w^2   0   0   0]
[  0   0   1   0   0   0]
[  0   0   0   1   0   0]
[  0   0   0   0   1   0]
[  0   0   0   0   0   1]
```

```
[w 1 1 0 0 0]
[1 1 0 0 0 0]
[1 0 0 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]

[  1   0   0   0   0   0]
[  0   1   0   0   0   0]
[  0   0   1   0   0   0]
[  0   0   0   1   w   w]
[  0   0   0   0   1 w^2]
[  0   0   0   0   0   1]

[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 1 0 0 0]
[0 0 0 w 1 1]
[0 0 0 1 1 0]
[0 0 0 1 0 0]
> Order(D);
46656
> H := SymmetricGroup(3);
> E := WreathProduct(G, H);
> Degree(E);
9
> Order(E);
60466176
> F := TensorWreathProduct(G, H);
> Degree(F);
27
> Order(F);
6718464
```

59.4 Operations on Matrices

59.4.1 Arithmetic with Matrices

`g * h`

The product of matrix g and matrix h, where g and h belong to the same generic group U. If g and h both belong to the same proper subgroup G of U, then the result will be returned as an element of G; if g and h belong to subgroups H and K of a subgroup G of U then the product is returned as an element of G. Otherwise, the product is returned as an element of U.

`g ^ n`

The n-th power of the matrix g, where n is a positive or negative integer.

`g / h`

The product of the matrix g by the inverse of the matrix h, i.e. the element $g * h^{-1}$. Here g and h must belong to the same generic group U. The rules for determining the parent group of g/h are the same as for $g * h$.

`g ^ h`

The conjugate of the matrix g by the matrix h, i.e. the element $h^{-1} * g * h$. Here g and h must belong to the same generic group U. The rules for determining the parent group of g^h are the same as for $g * h$.

`(g, h)`

The commutator of the matrices g and h, i.e. the element $g^{-1} * h^{-1} * g * h$. Here g and h must belong to the same generic group U. The rules for determining the parent group of (g, h) are the same as those for $g * h$.

`(g1, ..., gr)`

Given r matrices $g_1, \ldots, g_r$ belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

Example H59E7

These operations will be illustrated using the group GL$(3, 4)$.

```
> K<w> := FiniteField(4);
> GL34 := GeneralLinearGroup(3, K);
> x := GL34 ! [1,w,0, 0,w,1, w^2,0,1];
> y := GL34 ! [1,0,0, 1,w,0, 1,1,w];
> x;
[  1   w   0]
[  0   w   1]
[w^2   0   1]
> y;
[1 0 0]
[1 w 0]
```

```
[1 1 w]
> x*y;
[w^2 w^2   0]
[w^2   w   w]
[  w   1   w]
> x^10;
[  w   w   1]
[  w   1   1]
[  w w^2   w]
> x^-1;
[w^2 w^2 w^2]
[  1   w   w]
[  w   w w^2]
> x^y;
[w^2 w^2   0]
[  0 w^2   1]
[w^2 w^2   w]
> x/y;
[  0   1   0]
[  0 w^2 w^2]
[  w   w w^2]
> (x, y);
[  0   w   w]
[  w w^2   1]
[w^2   w w^2]
> (x,y,y);
[w^2   w w^2]
[w^2   w   0]
[w^2   1   w]
```

Arithmetic with group elements is not limited to elements of finite groups. We illustrate with a group of degree 3 over a function field.

```
> P<a,b,c,m,x,y,z> := FunctionField(RationalField(), 7);
> S := MatrixGroup< 3, P | [1,a,b,0,1,c,0,0,1],
>                          [1,0,m,0,1,0,0,0,1],
>                          [1,x,y,0,1,z,0,0,1] >;
>
> t := S.1 * S.2;
> t;
[    1     a b + m]
[    0     1     c]
[    0     0     1]
> t^-1;
[          1          -a a*c - b - m]
[          0           1          -c]
[          0           0           1]
> Determinant(t);
1
```

```
> t^2;
[              1             2*a a*c + 2*b + 2*m]
[              0               1             2*c]
[              0               0               1]
```

59.4.2 Predicates for Matrices

`g eq h`

Given matrices g and h belonging to the same generic group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given matrices g and h belonging to the same generic group, return `true` if g and h are distinct elements, `false` otherwise.

`IsIdentity(g)`

`IsId(g)`

Returns `true` if the matrix g is the identity matrix.

`IsScalar(g)`

Returns `true` if the matrix g is a scalar matrix.

59.4.3 Matrix Invariants

All of the functions for computing invariants of a square matrix apply to the elements of a matrix group. Here only operations of interest in the context of group elements are described. The reader is referred to chapter 26 for a complete list of functions applicable to matrices.

`Degree(g)`

The degree of the matrix g, i.e. the number of rows/columns of g.

`HasFiniteOrder(g)`

Returns `true` iff the matrix g has finite order. The second return value is the order if it is finite. The function rigorously proves its result (i.e., the result is not probable). Let R be the ring over which g is defined, and let the degree of the group in which g lies be n. If R is finite, then the first return value is trivially `true`.

If R is the integer ring then the function works as follows. Suppose first that g has finite order o. By a theorem of Minkowski (see Theorem 1.4 [KP02]), for any odd prime p, the reduction mod p of g has order o. Let $f(x) \in R[x]$ be the minimal polynomial of g. The matrix subalgebra generated by g is isomorphic to the quotient ring $R[x]/\langle f(x)\rangle$, so the order o of g equals the order of x mod $f(x)$.

For arbitrary g, the algorithm computes the order, $\bar{o}$, of the reduction of g modulo a small odd prime. If $\bar{o}$ is a possible order of an integer matrix of g's dimensions (see Theorem 2.7 *op. cit.*) then this is repeated with a larger prime. If this gives a different order, or the first attempt gave an impossible order, then g has infinite order. We now compute $x^{\bar{o}}$ mod $f(x)$. If this is 1, then $\bar{o}$ is the order of g, otherwise g has infinite order.

If R is the rational field then a necessary condition for g to have finite order is that $f(x)$ has integer coefficients, thus the above algorithm applies in this case.

If R is an algebraic number field of degree d over $\mathbf{Q}$ (including cyclotomic and quadratic fields), then the standard companion matrix blowup is applied to g to obtain a $(nd) \times (nd)$ matrix over $\mathbf{Q}$, and the above algorithm is then applied to this matrix.

`Order(g)`

`Proof` BoolElt *Default :* `true`

Given an element g of finite order belonging to a matrix group, this function returns the order of g. If g has infinite order, a runtime error results. In the case of a matrix group over a finite field, the algorithm described in [CLG97] is used. In all other cases, simple powering of g is used.

The parameter `Proof` is associated with the case when the coefficient ring for g is a finite field. In that case, if `Proof` is set to `false`, then difficult integer factorizations will not attempted. In this situation two values are returned of which the first is a multiple n of the order of g. and the second value indicates whether n is known to be the exact order of g.

`FactoredOrder(g)`

`Proof` BoolElt *Default :* `true`

Given an element g of finite order belonging to a matrix group, this function returns the order of g as a factored integer. If g has infinite order, a runtime error results. If g has infinite order, the function generates a runtime error. In the case of a matrix group over a finite field, the algorithm described in [CLG97] is used. In all other cases, simple powering of g is used. In that case it is more efficient to use this

function rather than factorizing the integer returned by `Order(g)`. If g has infinite order, an error ensues.

If the parameter `Proof` is `false`, then difficult integer factorizations are not attempted and the first return value F may contain composite numbers (so that the factorization expands to a multiple of the order of g); in any case the second return value indicates whether F is known to be the exact factored order of g.

`ProjectiveOrder(g)`

`Proof` BoolElt *Default :* `true`

The projective order n of the matrix g, and a scalar s such that $g^n = sI$. The projective order of g is the smallest n such that g^n is a scalar matrix (not just the identity matrix), and it always divides the true order of A. The parameter `Proof` is as for `Order`.

`FactoredProjectiveOrder(A)`

`Proof` BoolElt *Default :* `true`

Given a square invertible matrix A over a finite field K, return the projective order n of A in factored form and a scalar $s \in K$ such that $A^n = sI$. The parameter `Proof` is as for `FactoredOrder`.

`CentralOrder(g : parameters)`

`CentralOrder(g)`

`Proof` BoolElt *Default :* `true`

Return the smallest n such that g^n is central in its parent group. If g is a matrix and the optional parameter `Proof` is `false`, then accept a multiple of this value; the second value returned is `true` if the answer is exact.

`Determinant(g)`

The determinant of the matrix g.

`Trace(g)`

The trace of the matrix g.

`CharacteristicPolynomial(g: parameters)`

`Al` MonStgElt *Default :* "*Modular*"

`Proof` BoolElt *Default :* `true`

Given a matrix g belonging to a subgroup of $\mathrm{GL}(n, R)$, where R is a field or Euclidean Domain, return the characteristic polynomial of g as an element of the univariate polynomial ring over R. For details on the parameters, see the function `CharacteristicPolynomial` in the chapter on matrices.

`MinimalPolynomial(g)`

Given a matrix g belonging to a subgroup of $\mathrm{GL}(n, R)$, where R is a field or Z, return the minimal polynomial of g as an element of the univariate polynomial ring over R.

Example H59E8

We illustrate the matrix operations by applying them to some elements of $\mathrm{GL}(3, 4)$.

```
> K<w> := FiniteField(4);
> GL34 := GeneralLinearGroup(3, K);
> x := GL34 ! [w,0,1, 0,1,0, 1,0,1];
> x;
[w 0 1]
[0 1 0]
[1 0 1]
> Degree(x);
3
> Determinant(x);
w^2
> Trace(x);
w
> Order(x);
15
> m<t> := MinimalPolynomial(x);
> m;
t^3 + w*t^2 + w^2
> Factorization(m);
[
    <t + 1, 1>,
    <t^2 + w^2*t + w^2, 1>
]
> c<t> := CharacteristicPolynomial(x);
> c;
t^3 + w*t^2 + w^2
```

59.5 Global Properties

Unless otherwise noted, the functions in this section assume that a BSGS-representation for the group can be constructed.

59.5.1 Group Order

Unless the order is already known, each of the functions in this family will create a base and strong generating set for the group if one does not already exist.

`IsFinite(G)`

Given a matrix group G, return whether G is finite together with the order of G if G is finite. The function rigorously proves its result (i.e., the result is not probable). Let R be the ring over which G is defined, and let the degree of G be n. If R is finite, then the first return value is trivially `true`.

If R is the integer ring or rational field, then the function works as follows. The function successively generates random elements of G and tests whether each element has infinite order via the function `HasFiniteOrder`; if so, then the non-finiteness of G is proven. Otherwise, at regular intervals, the function attempts to construct a positive definite form fixed by G (see the function `PositiveDefiniteForm` in the chapter on matrix groups over $\mathbf{Q}$ and $\mathbf{Z}$), using a finite number of steps; if one is successively constructed, then the finiteness of G is proven. The number of steps attempted for the positive definite form constructed is increased as the algorithm progresses; if G is finite, such a form must exist and will be found when enough steps are tried, while if G is infinite, an element of infinite order is found very quickly in practice.

If R is an algebraic number field of degree d over $\mathbf{Q}$ (including cyclotomic and quadratic fields), then the standard companion matrix blowup is applied to the generators of G to obtain an isomorphic matrix group of (nd) over $\mathbf{Q}$, and the above algorithm is then applied to this matrix group.

`Order(G)`

`#G`

The order of the group G as an integer. If the order is not currently known, a base and strong generating set will be constructed for G. If G has infinite order, an error ensues.

`FactoredOrder(G)`

The order of the group G returned as a factored integer. The format is the same as for `FactoredIndex`. If the order of G is not known, it will be computed. If G has infinite order, an error ensues.

Example H59E9

```
> G := MatrixGroup<2,Integers()|[1,1,0,1],[0,1,-1,0]>;
> IsFinite(G);
false
> G24, e := ChangeRing(G, Integers(24));
> Order(G24);
9216
```

```
> G.-1*G.2;
[ 1  1]
[-1  0]
> (G.-1*G.2) @ e;
[ 1  1]
[23  0]
> (G24.2^2) @@ e;
[23  0]
[ 0 23]
```

59.5.2 Membership and Equality

`g in G`

Given a matrix g and a matrix group G, return `true` if g is an element of G, `false` otherwise.

`g notin G`

Given a matrix g and a matrix group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G`

Given a matrix group G and a set S of matrices belonging to a group H, where G and H belong to the same generic group, return `true` if S is a subset of G, `false` otherwise.

`H subset G`

Given matrix groups G and H belonging to the same generic group, return `true` if H is a subgroup of G, `false` otherwise.

`S notsubset G`

Given a matrix group G and a set S of matrices belonging to a group H, where G and H belong to the same generic group, return `true` if S is not a subset of G, `false` otherwise.

`H notsubset G`

Given matrix groups G and H belonging to the same generic group, return `true` if H is not a subgroup of G, `false` otherwise.

`H eq G`

Given matrix groups G and H belonging to the same generic group, return `true` if G and H are the same group, `false` otherwise.

`H ne G`

Given matrix groups G and H belonging to the same generic group, return `true` if G and H are distinct groups, `false` otherwise.

59.5.3 Set Operations

The creation of a base and strong generating set for a matrix group G provides us with a very compact representation of the set of elements of G. A particular BSGS imposes an order on the elements of G (lexicographic ordering of base images). It thus makes sense to talk about the 'number' of a group element relative to a particular BSGS.

`NumberingMap(G)`

A bijective mapping from the group G onto the set of integers $\{1 \dots |G|\}$. The actual mapping depends upon the base and strong generating set chosen for G.

`RandomProcess(G)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 20

Create a process to generate randomly chosen elements from the finite group G. The process is based on the product-replacement algorithm of [CLGM$^+$95], modified by the use of an accumulator. At all times, N elements are stored where N is the maximum of the specified value for `Slots` and `Ngens`$(G)+1$. Initially, these are just the generators of G. As well, one extra group element is stored, the accumulator. Initially, this is the identity. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call chooses one of the elements in the slots and multiplies it into the accumulator. The element in that slot is replaced by the product of it and another randomly chosen slot. The random value returned is the new accumulator value. Setting `Scramble := m` causes m such operations to be performed before the process is returned.

`Random(G: parameters)`

`Short`	BoolElt	*Default :* `false`

A randomly chosen element for the group G. If a BSGS is known for G, then the element chosen will be genuinely random. If no BSGS is known, then the *random* element is chosen by multiplying out a *random* word in the generators. Since it is not usually practical to choose words long enough to properly sample the elements of G, the element returned will usually be biased. The boolean-valued parameter `Short` is used in this situation to indicate that a short word will suffice. Thus, if `Random` is invoked with `Short` assigned the value `true` then the element is constructed using a short word.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the finite group G, construct a random element of G by forming a random product over the expanded generating set constructed when the process was created. For large degree groups, or groups for which a BSGS is not known, this function should be used in preference to `Random(G)`.

Example H59E10

We use the random function to sample the orders of elements in the group $\mathrm{GL}(20, 16)$.

```
> G := GeneralLinearGroup(20, GF(16));
> RP := RandomProcess(G);
> [ FactoredOrder(Random(RP)) : i in [1..20] ];
[
    [ <3, 1>, <5, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>,
    <61681, 1> ],
    [ <3, 1>, <5, 1>, <17, 1>, <23, 1>, <89, 1>, <257, 1>, <397, 1>, <683, 1>,
    <2113, 1> ],
    [ <3, 1>, <5, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>,
    <61681, 1> ],
    [ <3, 1>, <31, 1>, <8191, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>,
    <61681, 1> ],
    [ <3, 3>, <5, 1>, <7, 1>, <13, 1>, <17, 1>, <19, 1>, <29, 1>, <37, 1>,
    <43, 1>, <73, 1>, <109, 1>, <113, 1>, <127, 1>, <257, 1> ],
    [ <5, 1> ],
    [ <3, 1>, <5, 1> ],
    [ <3, 1>, <5, 2>, <11, 1>, <17, 1>, <31, 1>, <41, 1>, <53, 1>, <157, 1>,
    <1613, 1>, <2731, 1>, <8191, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <13, 1>, <17, 1>, <97, 1>, <241, 1>, <257, 1>,
    <673, 1> ],
    [ <3, 1>, <5, 1>, <17, 1>, <29, 1>, <43, 1>, <113, 1>, <127, 1>, <257, 1>,
    <65537, 1> ],
    [ <3, 1>, <5, 2>, <11, 1>, <29, 1>, <31, 1>, <41, 1>, <43, 1>, <113, 1>,
    <127, 1> ],
    [ <3, 1>, <5, 2>, <11, 1>, <17, 1>, <31, 1>, <41, 1>, <53, 1>, <157, 1>,
    <1613, 1>, <2731, 1>, <8191, 1> ],
    [ <3, 2>, <5, 2>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>, <61, 1>,
    <151, 1>, <257, 1>, <331, 1>, <1321, 1> ],
    [ <3, 1>, <5, 1>, <11, 1>, <31, 1>, <41, 1>, <257, 1>, <61681, 1>,
    <4278255361, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>,
    <61681, 1> ],
    [ <3, 1>, <5, 1>, <17, 1>, <23, 1>, <89, 1>, <257, 1>, <397, 1>, <683, 1>,
    <2113, 1> ], [ <3, 2>, <5, 1>, <7, 1>, <11, 1>, <13, 1>, <23, 1>, <31, 1>,
    <41, 1>, <89, 1>, <397, 1>, <683, 1>, <2113, 1> ]
]
```

59.6 Abstract Group Predicates

`IsAbelian(G)`

Returns `true` if the group G is abelian, `false` otherwise.

`IsCyclic(G)`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group G is elementary abelian, `false` otherwise.

`IsNilpotent(G)`

Returns `true` if the group G is nilpotent, `false` otherwise.

`IsSoluble(G)`
`IsSolvable(G)`

Returns `true` if the group G is soluble, `false` otherwise.

`IsPerfect(G)`

Returns `true` if the group G is perfect, `false` otherwise.

`IsSimple(G)`

Returns `true` if the group G is simple, `false` otherwise.

Example H59E11

We illustrate the functions of the last two section by applying them to a group of degree 6 over the field $\mathbf{F}_9$.

```
> F9<w> := GF(9);
> y := w^6;  z := w^2;
> J2A2 := MatrixGroup< 6, F9 | [y, 1-y, z,0,0,0, 1-y ,z, -1,0,0,0, z, -1,1+y,
>                               0,0,0,0,0,0, z, 1+y, y, 0,0,0,1+y, y, -1, 0,
>                               0,0, y ,-1,1-y],
>                              [1+y, z, y, 0,0,0, z, 1+y, z, 0,0,0, y, z, 1+y,
>                               0,0,0, z, 0,0,1-y, y, z, 0, z, 0, y, 1-y, y,
>                               0,0, z, z, y, 1-y],
>                              [0,0,0,y, 0,0, 0,0,0,0,y, 0, 0,0,0,0,0,y,
>                               y, 0,0,0,0,0, 0,y, 0,0,0,0, 0,0,y, 0,0,0] >;
> J2A2;
MatrixGroup(6, GF(3, 2))
Generators:
[w^6 w^3 w^2   0   0   0]
[w^3 w^2   2   0   0   0]
[w^2   2   w   0   0   0]
[  0   0   0 w^2   w w^6]
[  0   0   0   w w^6   2]
```

```
[  0   0   0 w^6   2 w^3]

[  w w^2 w^6   0   0   0]
[w^2   w w^2   0   0   0]
[w^6 w^2   w   0   0   0]
[w^2   0   0 w^3 w^6 w^2]
[  0 w^2   0 w^6 w^3 w^6]
[  0   0 w^2 w^2 w^6 w^3]

[  0   0   0 w^6   0   0]
[  0   0   0   0 w^6   0]
[  0   0   0   0   0 w^6]
[w^6   0   0   0   0   0]
[  0 w^6   0   0   0   0]
[  0   0 w^6   0   0   0]
> Order(J2A2);
1209600
> FactoredOrder(J2A2);
[ <2, 8>, <3, 3>, <5, 2>, <7, 1> ]
> IsSoluble(J2A2);
false
> IsPerfect(J2A2);
true
> IsSimple(J2A2);
false
```

Thus the group is non-soluble and perfect but it is not a simple group. We examine its Sylow2-subgroup.

```
> S2 := SylowSubgroup(J2A2, 2);
> IsAbelian(S2);
false
> IsNilpotent(S2);
true
> IsSpecial(S2);
false
```

59.7 Conjugacy

`Class(H, x)`

`Conjugates(H, x)`

Given a group H and an element x belonging to a group K such that H and K are subgroups of the same general linear group, this function returns the set of conjugates of x under the action of H. If $H = K$, the function returns the conjugacy class of x in H.

`ClassMap(G)`

Given a group G, construct the conjugacy classes and the class map f for G. For any element x of G, $f(x)$ will be the conjugacy class representative chosen by the `Classes` function.

`ConjugacyClasses(G: parameters)`

`Classes(G: parameters)`

`WeakLimit`	RngIntElt	*Default :* 500
`StrongLimit`	RngIntElt	*Default :* 5000
`Al`	MonStgElt	*Default :*

Construct a set of representatives for the conjugacy classes of the matrix group G. The classes are returned as a sequence of triples containing the element order, the class length and a representative element for the class. The parameter `Al` enables the user to select the algorithm that is to be used.

`Al := "Action"`: Create the classes of G by computing the orbits of the set of elements of G under the action of conjugation. This option is only feasible for small groups.

`Al := "Random"`: Construct the conjugacy classes of elements for a matrix group G using an algorithm that searches for representatives of all conjugacy of G by examining a random selection of group elements and their powers. The behaviour of this algorithm is controlled by two associated optional parameters `WeakLimit` and `StrongLimit`, whose values are positive integers n_1 and n_2, say. Before describing the effect of these parameters, some definitions are needed: A mapping $f : G \to I$ is called a class invariant if $f(g) = f(g^h)$ for all $g, h \in G$. In matrix groups, the primary invariant factors are used where possible, or the characteristic or minimal polynomials otherwise. Two matrices g and h are said to be *weakly conjugate* with respect to the class invariant f if $f(g) = f(h)$. By definition, conjugacy implies weak conjugacy, but the converse is false. The random algorithm first examines n_1 random elements and their powers, using a test for weak conjugacy. It then proceeds to examine a further n_2 random elements and their powers, using a test for ordinary conjugacy. The idea behind this strategy is that the algorithm should attempt to find as many classes as possible using the very cheap test for weak conjugacy, before employing the more expensive ordinary conjugacy test to recognize the remaining classes.

`Al := "Extend"`: Construct the conjugacy classes of G by first computing classes in a quotient G/N and then extending these classes to successively larger quotients G/H until the classes for $G/1$ are known. More precisely, a series of subgroups $1 = G_0 < G_1 < \cdots < G_r = R < G$ is computed such that R is the (solvable) radical of G and G_{i+1}/G_i is elementary abelian. The radical quotient G/R is computed and its classes and centralizers of their representatives found using the permutation group algorithm, and pulled back to G. The parameters `TFAl` and `ASAl` control the algorithm used to compute the classes of G/R. See the `GrpPerm` chapter for more information on these parameters.

To extend from G/G_{i+1} to the next larger quotient G/G_i, an affine action of each centralizer on a quotient of the elementary abelian layer G_{i+1}/G_i is computed. Each distinct orbit in that action gives rise to a new class of the larger quotient (see Mecky and Neubuser [MN89]).

`Al := "Lifting"`: Construct a permutation representation for G, compute the classes of the representation, and lift them back to G through the kernel of the representation. Successful when the kernel is small. Currently uses the permutation action of G on its first basic orbit as the permutation representation.

`Al := "Classic"`: Construct the conjugacy classes by enumeration of class invariants. This algorithm is only available for classical groups. It has only been implemented for groups containing the special linear group and for the conformal unitary group.

Default: The classic algorithm will be used if G is recognised to contain the special linear group (using `IsLinearGroup`), or if G is known to be conformal unitary group in the standard representation (that is, if G was constructed by `ConformalUnitaryGroup`). The action algorithm will be used if $|G| \leq 2000$. If G is soluble then classes are computed in a PC-representation of G. When $|G| > 2000$ and the base ring of G is a finite field then the Extension algorithm is used. Otherwise the Lifting algorithm is used, unless the kernel size exceeds 10000. If there is a big kernel and the base ring of the group can be embedded in a field then the extension algorithm is used. Otherwise the random algorithm will be applied with the limits given by the parameters `WeakLimit` and `StrongLimit`. If that fails to compute all the classes and $|G| \leq 100000$, then the action algorithm will be used.

`ClassRepresentative(G, x)`

Given a group G for which the conjugacy classes are known and an element x of G, return the designated representative for the conjugacy class of G containing x.

`ClassCentraliser(G, i)`

The centraliser of the representative element stored for conjugacy class number i in group G. The group computed is stored with the class table for reference by future calls to this function.

`ClassInvariants(G, g)`

`ClassInvariants(G, i)`

The invariants for the conjugacy class of g in G or the conjugacy class number i in G. The type of invariants may vary depending on the group. This is only available for groups, for which the *classic* algorithm for computing conjugacy classes is available.

`ClassRepresentativeFromInvariants(G, p, h, t)`

Given a group G, for which the *classic* algorithm for computing conjugacy classes is available, and the class invariants p, h and t, return the standard class representative for the conjugacy class in G with the given invariants.

`IsConjugate(G, g, h)`

Given a group G and elements g and h belonging to G, return the value `true` if g and h are conjugate in G. The function returns a second value in the event that the elements are conjugate: an element k which conjugates g into h.

`IsConjugate(G, H, K)`

Given a group G and subgroups H and K belonging to G, return the value `true` if H and K are conjugate in G. The function returns a second value in the event that the subgroups are conjugate: an element z which conjugates H into K.

`IsGLConjugate(H, K)`

Given H and K, both subgroups of the same general linear group $G = GL_n(q)$, return the value `true` if H and K are conjugate in G. The function returns a second value in the event that the subgroups are conjugate: an element z which conjugates H into K. The algorithm is described in Roney-Dougal [RD04].

`Exponent(G)`

The exponent of the group G.

`NumberOfClasses(G)`

`Nclasses(G)`

The number of conjugacy classes of elements for the group G.

`PowerMap(G)`

Given a group G, construct the power map for G. Suppose that the order of G is m and that G has r conjugacy classes. When the classes are determined by MAGMA, they are numbered from 1 to r. Let C be the set of class indices $\{1, \ldots, r\}$ and let P be the set of integers $\{1, \ldots, m\}$. The *power map* f for G is the mapping,

$$f : C \times P \to C$$

where the value of $f(i, j)$ for $i \in C$ and $j \in P$ is the number of the class which contains x_i^j, where x_i is a representative of the i-th conjugacy class.

`AssertAttribute(G, "Classes", Q)`

Given a group G, and a sequence Q of k distinct elements of G, one from each conjugacy class, use Q to define the classes attribute of G. The sequence Q may be either a sequence of elements of G or, preferably, a sequence of pairs `<GrpMatElt, RngIntElt>` giving class representatives and their class length. In this latter case, no backtrack searches are performed.

Example H59E12

We take a group from the database of rational matrix groups and compute its conjugacy classes. The group has degree 12 and is written over the integers.

```
> DB := RationalMatrixGroupDatabase();
> G := Group(DB, 12, 3);
> FactoredOrder(G);
[ <2, 17>, <3, 8>, <5, 2> ]
> CompositionFactors(G);
    G
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  C(2, 3)               = S(4, 3)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  C(2, 3)               = S(4, 3)
    *
    |  Cyclic(2)
    1
```

The conjugacy classes of `G` are computed as follows:

```
> time cl := Classes(G);
Time: 18.580
> #cl;
1325
```

The group has 1325 conjugacy classes of elements.

59.8 Subgroups

59.8.1 Construction of Subgroups

`sub< G | L >`

Given the matrix group G, construct the subgroup H of G generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) A sequence of n integers defining a matrix of G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic matrix group. The subgroup H will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of H consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list. Repetitions of an element and occurrences of the identity element are removed.

`ncl< G | L >`

Given the matrix group G, construct the subgroup H of G that is the *normal closure* of the subgroup H generated by the elements specified by the list L, where the possibilities for L are the same as for the `sub`-constructor.

Example H59E13

We define $O^-(4,2)$ as a subgroup of $\mathrm{GL}(4,2)$. Recall that $O^-(4,2)$ is isomorphic to S_5. We then locate a subset of its generators that lie within the subgroup isomorphic to A_5.

```
> GL42 := GeneralLinearGroup(4, GF(2));
> Ominus42 := sub< GL42 | [1,0,0,0, 1,1,0,1, 1,0,1,0, 0,0,0,1 ],
>                         [0,1,0,0, 1,0,0,0, 0,0,1,0, 0,0,0,1 ],
>                         [0,1,0,0, 1,0,0,0, 0,0,1,0, 0,0,1,1 ] >;
> Order(Ominus42);
120
> H := sub< Ominus42 | $.1, $.3 >;
print Order(H);
10
> N := ncl< Ominus42 | $.1, $.3 >;
> Order(N);
60
```

59.8.2 Elementary Properties of Subgroups

`Index(G, H)`

The index of the subgroup H in the group G. The index is returned as an integer. If the orders of G and H are not known, they will be computed.

`FactoredIndex(G, H)`

The index of the subgroup H in the group G. The index is returned as a factored integer. The factorization is returned in the form of a sequence Q which is defined as follows: If $[G : H] = p_1^{e_1} \dots p_n^{e_n}, \quad e_i \neq 0$, then Q will be the integer sequence $[< p_1, e_1 >, \dots, < p_n, e_n >]$. If the orders of G and H are not known, they will be computed.

`IsCentral(G, H)`

Returns `true` if the subgroup H of the group G lies in the centre of G, `false` otherwise.

`IsMaximal(G, H)`

Returns `true` if the subgroup H of the group G is a maximal subgroup of G. This function is evaluated by constructing the permutation representation of G on the cosets of H and testing this representation for primitivity. For this reason, the use of `IsMaximal` should be avoided if the index of H in G exceeds a few thousand.

`IsNormal(G, H)`

Returns `true` if the subgroup H of the group G is a normal subgroup of G, `false` otherwise.

`IsSubnormal(G, H)`

Returns `true` if the subgroup H of the group G is subnormal in G, `false` otherwise.

59.8.3 Standard Subgroups

`H ^ g`

`Conjugate(H, g)`

Construct the conjugate $g^{-1} * H * g$ of the matrix group H by the matrix g. The group H and the element g must belong to a common matrix group.

`H meet K`

Given groups H and K which belong to the same matrix group, construct the intersection of H and K.

`CommutatorSubgroup(G, H, K)`

`CommutatorSubgroup(H, K)`

Given subgroups H and K of the group G, construct the commutator subgroup of H and K as a subgroup of G. If K is a subgroup of H, then G may be omitted.

`Centralizer(G, g)`

Construct the centralizer of the matrix g in the group G; g and G must belong to a common matrix group.

`Centralizer(G, H)`

Construct the centralizer of the group H in the group G; G and H must belong to a common matrix group.

`Core(G, H)`

Given a subgroup H of the matrix group G, construct the maximal normal subgroup of G that is contained in the subgroup H.

`H ^ G`

`NormalClosure(G, H)`

Given a subgroup H of the matrix group G, construct the normal closure of H in G.

`Normalizer(G, H)`

Given a subgroup H of the group G, construct the normalizer of H in G.

`SylowSubgroup(G, p)`

`Sylow(G, p)`

Given a group G and a prime p, construct the Sylow p-subgroup of G.

`pCore(G, p)`

Given a group G and a prime p dividing the order of G, construct the maximal normal p-subgroup of G.

59.8.4 Low Index Subgroups

`LowIndexSubgroups(G, R : parameters)`

`LowIndexSubgroups(G, R: parameters)`

Given a matrix group G, and an expression R defining a positive integer range (see below), determine the conjugacy classes of subgroups of G whose indices lie in the range specified by R. The subgroups are returned as a sequence of subgroups of G. The argument R is one of the following:

(a) An integer n representing the range $[1, n]$;

(b) A tuple $< a, b >$ representing the range $[a, b]$.

The subgroups are constructed using an algorithm due to Leedham-Green & O'Brien [LGO02]. In practice, the algorithm is most useful for small values of n, say up to 8.

The algorithm proceeds by iteratively constructing better approximations to finite presentations for G/K, where K is the intersection of kernels of all homomorphisms from G into S_n, and applying `LowIndexSubgroups` to the resulting finitely-presented group. The output information displayed for various values of the `Print` parameter about the number and existence of putative subgroups of index at most n refers to the current finite presentation only, may change as this presentation is further refined, and need not be reflected in the final answer.

`Limit` RngIntElt *Default :* ∞

Terminate after finding n conjugacy classes of subgroups satisfying the designated conditions.

`Print` RngIntElt *Default :* 0

The `Print` parameter takes values from 0 to 3. The information displayed

Example H59E14

```
> G := GL (4, 5);
> L := LowIndexSubgroups (G, 4);
> #L;
3
> L[3];
MatrixGroup(4, GF(5))
Generators:
    [4 0 0 4]
    [1 0 0 0]
    [0 4 0 0]
    [0 0 4 0]

    [4 0 0 3]
    [3 0 0 0]
    [0 4 0 0]
```

```
[0 0 4 0]

[4 0 0 1]
[4 0 0 0]
[0 4 0 0]
[0 0 4 0]

[4 0 0 2]
[2 0 0 0]
[0 4 0 0]
[0 0 4 0]
```

59.8.5 Conjugacy Classes of Subgroups

`SubgroupClasses(G: parameters)`

`Subgroups(G: parameters)`

Representatives for the conjugacy classes of subgroups for the group G. The subgroups are returned as a sequence of records where the i-th record contains:

(a) A representative subgroup H for the i-th conjugacy class (field name `subgroup`).

(b) The order of the subgroup (field name `order`).

(c) The number of subgroups in the class (field name `length`).

(d) [Optional] A presentation for H (field name `presentation`).

`Al` MonStgElt *Default :* "*All*"

`Al := "All"`: Construct all subgroups of G.
`Al := "Maximal"`: Only construct maximal subgroups of G. This option reduces the number of intersections with any elementary abelian layer that need be considered and eliminates the need to recursively apply the algorithm.
`Al := "Normal"`: Only construct normal subgroups of G. This option does not use database lookup to find the normal subgroups of the radical quotient of G and also reduces the number of intersections with any layer that need be considered.

`LayerSizes` SeqEnum *Default :* See below

`LayerSizes := [ 2, 5, 3, 4, 7, 3, 11, 2, 17, 1]` is equivalent to the default. When constructing an Elementary Abelian series for the group, attempt to split 2-layers of size gt 2^5, 3-layers of size gt 3^4, etc. The implied exponent for 13 is 2 and for all primes greater than 17 the exponent is 1.

`Series` SeqEnum *Default :* See below

Use the given elementary abelian series rather than constructing the default series. The first subgroup in the series must be the solvable radical of G. The subgroups must form a descending chain of normal subgroups of G, such that each quotient

is elementary abelian. The last subgroup in the series must be either elementary abelian or trivial.

`Presentation` BoolElt *Default :* `false`

`Presentation := true`: Construct a presentation for each subgroup.

`OrderEqual` RngIntElt *Default :*

`OrderEqual := n`: Only construct subgroups having order equal to n.

`OrderDividing` RngIntElt *Default :*

`OrderDividing := n`: Only construct subgroups having order dividing n.

`OrderMultipleOf` RngIntElt *Default :*

`OrderMultipleOf := n`: Only construct subgroups having order a multiple of n.

`IndexLimit` RngIntElt *Default :*

`IndexLimit := n`: Only construct subgroups having index in G less than or equal to n.

`IsElementaryAbelian` BoolElt *Default :* `false`

`IsElementaryAbelian := true`: Only construct elementary abelian subgroups of G.

`IsCyclic` BoolElt *Default :* `false`

`IsCyclic := true`: Only construct cyclic subgroups of G.

`IsAbelian` BoolElt *Default :* `false`

`IsAbelian := true`: Only construct abelian subgroups of G.

`IsNilpotent` BoolElt *Default :* `false`

`IsNilpotent := true`: Only construct nilpotent subgroups of G.

`IsSolvable` BoolElt *Default :* `false`

`IsSolvable := true`: Only construct solvable subgroups of G.

`IsNotSolvable` BoolElt *Default :* `false`

`IsNotSolvable := true`: Only construct insolvable subgroups of G.

`IsPerfect` BoolElt *Default :* `false`

`IsPerfect := true`: Only construct perfect subgroups of G.

The Algorithm: (See Cannon, Cox and Holt [CCH01]) This command proceeds by first constructing an elementary abelian series for G together with G's radical quotient Q as a permutation group. (Thus this function is limited to matrix groups over fields, where the group has a BSGS.) The required subgroups of Q are then found as for permutation groups. We first attempt to locate the quotient in a database of groups with trivial Fitting subgroup. This database contains all such groups of order up to 216 000, and all such which are perfect of order up to 1 000 000. If Q is found then either all its subgroups, or its maximal subgroups are read from the database. (In some cases only the maximal subgroups are stored.) If Q is

not found then we attempt to find the maximal subgroups of Q using a method of Derek Holt. For this to succeed all simple factors of the socle of Q must be found in a second database which currently contains all simple groups of order less than 1.6×10^7, as well as M_{24}, HS, J_3, McL, $Sz(32)$ and $L_6(2)$. There are also special routines to handle numerous other groups. These include: A_n for $n \leq 999$, $L_2(q)$, $L_3(q)$, $L_4(q)$ and $L_5(q)$ for all q, $U_3(q)$ for q prime and $q = 8, 9, 16, 25$, $U_4(q)$ for $q = 4, 5, 7$, $S_4(q)$ for all odd q and even $q \leq 16$, $L_d(2)$ for $d \leq 14$, and the following groups: $L_6(3)$, $L_7(3)$, $U_6(2)$, $S_8(2)$, $S_{10}(2)$, $O_8^{\pm}(2)$, $O_{10}^{\pm}(2)$, $S_6(3)$, $O_7(3)$, $O_8^-(3)$, $G_2(4)$, $G_2(5)$, ${}^3D_4(2)$, ${}^2F_4(2)'$, Co_2, Co_3, He, Fi_{22}.

If we have only maximal subgroups of Q, and more are required, we apply the algorithm recursively to the maximal subgroups to determine all subgroups of Q. This may take some time.

The subgroups of Q are then pulled back to G and extended to the whole group by stepwise extension through each layer of the elementary abelian series. For each layer this involves determining all possible intersections of a subgroup with this layer and all extensions with this intersection.

The limitations are that the simple factors of the socle of Q must be in the list above. Further, it may take some time to construct all subgroups from the maximal subgroups first found, and, if there is a large elementary abelian layer, there will be many possible intersections, which could also make the algorithm prohibitively slow.

There are numerous parameters for this function which allow the user to place restrictions on which subgroup classes are constructed. Using these restrictions may help overcome the problems noted above.

`MaximalSubgroups(G:` *parameters*`)`

Construct the sequence of maximal subgroup classes of the matrix group G. This is equivalent to the command `Subgroups(G: Al := "Maximal")`. The same parameters as for `Subgroups` are available to limit the search.

`SubgroupsLift(G, A, B, Q:` *parameters*`)`

This function isolates one step of the extension process used by the `Subgroups` family of functions. Q is a sequence of records such as returned by `Subgroups(G)`. A and B are normal subgroups of G with A/B elementary abelian. The records in Q are interpreted as subgroups of G/A, which are lifted to all possible corresponding subgroups of G/B, subject to the parameters given.

59.9 Quotient Groups

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

59.9.1 Construction of Quotient Groups

`quo< G | L >`

Given the matrix group G, construct the quotient group $Q = G/N$, where N is the normal closure of the subgroup of G generated by the elements specified by L. The clause L is a list of one or more items of the following types:

(a) A sequence of n integers defining a matrix of G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic matrix group. The function returns

(a) the quotient group Q, and

(b) the natural homomorphism $f : G \to Q$.

Currently in MAGMA, the quotient group is constructed via the regular representation of the quotient, so that the application of this operator is restricted to the case where the index of N in G is small. The representation of the quotient that is returned is the result of applying degree reduction to the regular representation, so need not be regular.

The generators of the quotient group correspond to the generators of G.

`G / N`

Given a normal subgroup N of the matrix group G, construct the quotient of G by N. Currently in MAGMA, the quotient group is constructed via the regular representation of the quotient, so the application of this operator is restricted to the case where the index of N in G is small. The representation of the quotient that is returned is the result of applying degree reduction to the regular representation, so need not be regular.

Example H59E15

We determine the structure of a quotient in a soluble subgroup of $\mathrm{GL}(3, 5)$.

```
> G := MatrixGroup< 3, GF(5) | [0,1,0, 1,0,0, 0,0,1], [0,1,0, 0,0,1, 1,0,0 ],
>                                    [2,0,0, 0,1,0, 0,0,1] >;
> Order(G);
384
> Q, f := quo< G | G.2 >;
> Q;
Permutation group Q of degree 8
    (1, 2)(3, 4)(5, 6)(7, 8)
```

```
    Id(Q)
    (1, 3, 5, 7)(2, 4, 6, 8)
> IsAbelian(Q);
true
> AbelianInvariants(Q);
[ 4, 2 ]
```

59.9.2 Abelian, Nilpotent and Soluble Quotients

A number of standard quotients may be constructed. The method first constructs a presentation for the matrix group and then applies the appropriate fp-group algorithm.

`AbelianQuotient(G)`

The maximal abelian quotient G/G' of the group G as `GrpAb` (cf. chapter 69). The natural epimorphism $\pi : G \to G/G'$ is returned as second value.

`ElementaryAbelianQuotient(G, p)`

The maximal p-elementary abelian quotient Q of the group G as `GrpAb` (cf. chapter 69). The natural epimorphism $\pi : G \to Q$ is returned as second value.

`pQuotient(G, p, c)`

Given a matrix group G, a prime p and a positive integer c, construct a pc-presentation for the largest p-quotient P of G having lower exponent-p class at most c. If c is given as 0, then the limit 127 is placed on the class.

The function also returns the natural homomorphism π from G to P, a sequence S describing the definitions of the pc-generators of P and a flag indicating whether P is the maximal p-quotient of G.

The k-th element of S is a sequence of two integers, describing the definition of the k-th pc-generator $P.k$ of P as follows.

- If $S[k] = [0, r]$, then $P.k$ is defined via the image of $G.r$ under π.
- If $S[k] = [r, 0]$, then $P.k$ is defined via the power relation for $P.r$.
- If $S[k] = [r, s]$, then $P.k$ is defined via the conjugate relation involving $P.r^{P.s}$.

`NilpotentQuotient(G, c)`

This function returns the class c nilpotent quotient of the matrix group G, together with the epimorphism π from G onto this quotient.

`SolvableQuotient(G)`

`SolubleQuotient(G)`

The function returns the largest soluble quotient S of the matrix group G together with the epimorphism $\pi : G \to S$.

PCGroup(G)

For a solvable group G, the function returns an isomorphic group of type GrpPC together with an isomorphism from G to the new group. If G is not solvable, then the call to PCGroup will result in an error.

Example H59E16

We take a degree 10 matrix group over the integers and compute its maximal abelian and soluble quotients. The epimorphisms supplied by these two functions may be used to pass between the group and its quotients.

```
> DB := RationalMatrixGroupDatabase();
> G := Group(DB, 10, 2);
> G : Minimal;
MatrixGroup(10, Integer Ring) of order 4147200
> A := AbelianQuotient(G); A;
Abelian Group isomorphic to Z/2 + Z/2 + Z/2
Defined on 3 generators
Relations:
  2*A.1 = 0
  2*A.2 = 0
  2*A.3 = 0
> S, f := SolubleQuotient(G); S;
GrpPC : S of order 32 = 2^5
PC-Relations:
  S.2^2 = S.4,
  S.2^S.1 = S.2 * S.4,
  S.3^S.2 = S.3 * S.4 * S.5
> G.1 @ f;
S.1 * S.4 * S.5
> S.5 @@ f in DerivedGroup(G);
true
```

59.10 Matrix Group Actions

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

59.10.1 Orbits and Stabilizers

Let G be a matrix group and let M be its natural module. Now G has an action on the elements and submodules of M. A derived G-set for G consists of the closure under the natural action of G of one of the following:

- A set of vectors of M;
- A set of k element subsets of vectors of M;
- A set of k element sequences of vectors of M;
- A set of submodules of M, each of which has fixed dimension k;
- A cartesian product of G-sets.

`u * g`

Given an element g belonging to the matrix group G with natural module M and an element u of this module, return the vector $u * g$.

`y ^ g`

Given an element g belonging to the matrix group G with natural module M and an object y which is an element of some derived G-set of M, find the image of y under g.

`y ^ G`

`Orbit(G, y)`

Given a matrix group G with natural module M and an object y which is either a vector of M, a submodule of M, or a tuple whose components are either vectors or submodules, find the orbit of y under G.

`OrbitBounded(G, y, b)`

Given a matrix group G with natural module M and an object y which is either a vector of M, a submodule of M, or a tuple whose components are either vectors or submodules, return `true` if the orbit of y under G has length less than or equal to b. Otherwise the function returns `false`. If it returns `true`, then the orbit of y is returned as the second value.

`Orbits(G)`

Given a matrix group G with natural R-module M, construct the orbits of G on the vectors of M. The orbits are returned as a sequence of sets.

`LineOrbits(G)`

Given a matrix group G with natural R-module M, construct the orbits of G on the rank-1 submodules of M. The orbits are returned as a sequence of sets.

`OrbitClosure(G, S)`

Given a matrix group G with natural module M and a set S of vectors or subspaces of M, return the union of orbits of the elements of S under the natural action of G on M.

`Stabilizer(G, y)`

Given a matrix group G with natural module M and an object y which is either a vector of M, a submodule of M, or a tuple whose components are either vectors or submodules, determine the stabilizer of y in G.

Example H59E17

We continue with the group $J2A2$ introduced above.

```
> V := RSpace(G);
> u := V![1,0,0,0,0,0];
> U := sub< V | u >;
> x := < u, U >;
> W := sub< V | u, u*G.1 >;
> u^G.1;
(w^6 w^3 w^2   0   0   0)
> U^G.1;
Vector space of degree 6, dimension 1 over GF(3, 2)
Echelonized basis:
(  1 w^5   2   0   0   0)
> W^G.1;
Vector space of degree 6, dimension 2 over GF(3, 2)
Echelonized basis:
(  1 w^5   0   0   0   0)
(  0   0   1   0   0   0)
> x^G.1;
<(w^6 w^3 w^2   0   0   0), Vector space of degree 6,
dimension 1 over GF(3, 2)
Echelonized basis:
(  1 w^5   2   0   0   0)>
> H := sub< G | G.1, G.2 >;
> #Orbit(H, u);
252
> #Orbit(H, U);
63
> #Orbit(G, U);
3150
> Stabilizer(G, U);
MatrixGroup(6, GF(3^2)) of order 384 = 2^7 * 3
Generators:
[  2   0   0   0   0   0]
[w^3   w   w   0   2 w^2]
[w^5 w^7 w^7   0   1 w^2]
```

```
[  0   0   1   2   1   0]
[w^7 w^5   0   0   0 w^6]
[  w w^3   0   0   0 w^6]

[w^2   0   0   0   0   0]
[w^5 w^5 w^5   0   w   0]
[w^7 w^3 w^3   0   0 w^7]
[w^2 w^3   w w^6   w w^3]
[w^3   1 w^6   0   w w^7]
[  w w^6   2   0   w w^7]

[w^6   0   0   0   0   0]
[  0   2   0   0   0   0]
[  0   0 w^6   0   0   0]
[w^2 w^7 w^6 w^2   0   0]
[  w   0   w   0   2   0]
[w^6 w^7 w^2   0   0 w^2]

[  2   0   0   0   0   0]
[  0   2   0   0   0   0]
[  0   0   2   0   0   0]
[  0   0   0   2   0   0]
[  0   0   0   0   2   0]
[  0   0   0   0   0   2]
> #Orbit(H, x);
252
> #Orbit(H, W);
28
```

59.10.2 Orbit and Stabilizer Functions for Large Groups

In this section we describe a number of constructions for orbits and stabilizers which in certain circumstances may be applicable to much larger groups than the functions described above.

`OrbitsOfSpaces(G, k)`

Determine representatives and lengths for the orbits of all k-dimensional subspaces of the natural vector space under action of a matrix group defined over a prime field; return a sequence of tuples each containing an orbit length and representative. This function is very space-efficient and hence has a significantly larger range than the general-purpose `Orbits`; however, only representatives and lengths are stored. Theoretical details of the algorithm used may be found in O'Brien [O'B90].

`NumberOfFixedSpaces(x, s)`

`NumberOfFixedSpaces(x, s)`

Return number of subspaces of dimension s fixed by matrix x.

Example H59E18

```
> G := GL (4, 5);
> H := ExteriorSquare (G);
> H;
MatrixGroup(6, GF(5))
Generators:
    [2 0 0 0 0 0]
    [0 2 0 0 0 0]
    [0 0 1 0 0 0]
    [0 0 0 2 0 0]
    [0 0 0 0 1 0]
    [0 0 0 0 0 1]

    [0 0 0 1 0 0]
    [1 0 0 0 1 0]
    [1 0 0 0 0 0]
    [0 1 0 0 0 1]
    [0 1 0 0 0 0]
    [0 0 1 0 0 0]
> O := OrbitsOfSpaces (H, 2);
```

We see that there are four orbits:

```
> O;
[
    <
        4836,
        Vector space of degree 6, dimension 2 over GF(5)
        Generators:
        (1 0 0 0 0 0)
        (0 1 0 0 0 0)
        Echelonized basis:
        (1 0 0 0 0 0)
        (0 1 0 0 0 0)
    >,
    <
        96720,
        Vector space of degree 6, dimension 2 over GF(5)
        Generators:
        (1 0 1 1 0 0)
        (0 1 0 0 0 0)
        Echelonized basis:
        (1 0 1 1 0 0)
        (0 1 0 0 0 0)
    >,
    <
        251875,
        Vector space of degree 6, dimension 2 over GF(5)
```

```
        Generators:
        (1 0 0 0 1 0)
        (0 1 0 0 0 0)
        Echelonized basis:
        (1 0 0 0 1 0)
        (0 1 0 0 0 0)
    >,
    <
        155000,
        Vector space of degree 6, dimension 2 over GF(5)
        Generators:
        (1 0 1 1 1 0)
        (0 1 1 1 0 0)
        Echelonized basis:
        (1 0 1 1 1 0)
        (0 1 1 1 0 0)
    >
]
```

We compute the number of spaces of dimension 2 fixed by $H.1$ and the number of spaces of dimension 3 fixed by $H.2$.

```
> NumberOfFixedSpaces(H.1, 2);
1023
> NumberOfFixedSpaces(H.2, 3);
2
```

`EstimateOrbit(G, v: parameters)`

`EstimateOrbit(G, U: parameters)`

`MaxSize`	RngIntElt	*Default :* 10000
`NumberCoincidences`	RngIntElt	*Default :* 15

Estimate the size of the orbit of the vector v or subspace U of natural vector space under the action of matrix group G by constructing at most `MaxSize` random elements of the orbit and counting at most `NumberCoincidences` coincidences. The function returns a lower bound, upper bound, and estimate of size; if insufficient coincidences are found to estimate the orbit size, the function returns 0. Theoretical details of the algorithm used may be found in Eick, Leedham-Green and O'Brien [ELGO02].

`ApproximateStabiliser(G, A, U: parameters)`

`ImageGenerators`	SeqEnum	*Default :* []
`MaxSize`	RngIntElt	*Default :* 10000
`NumberCoincidences`	RngIntElt	*Default :* 15
`OrderCheck`	BoolElt	*Default :* `false`

A is image of representation of G and A acts on U, a subspace or vector. Approximate the stabiliser of U under A. We assume either a $1-1$ correspondence between generators of G and those of A, or between generators of G and those elements of A supplied as `ImageGenerators`. Elements of G whose images in A fix U are obtained by constructing at most `MaxSize` elements of the orbit of U under A or until we find `Numbercoincidences` repetitions in this orbit; if `OrderCheck` is `true`, report the order of the subgroup S of A which is found. Return preimage of S in G and S, together with a lower bound, upper bound, and estimate of the size of orbit of U. If insufficient coincidences are found to estimate the orbit size, the function returns these last values as 0.

Example H59E19

```
> G := GL (4, 5);
> A := ExteriorSquare (G);
> V := VectorSpace (GF (5), 6);
> U := sub < V | [Random (V): i in [1..2]]>;
> U;
Vector space of degree 6, dimension 2 over GF(5)
Generators:
(4 3 2 1 0 2)
(3 2 2 4 4 1)
Echelonized basis:
(1 0 2 0 2 4)
(0 1 3 2 4 2)
> EstimateOrbit (A, U);
209316 594421 324272
> H, B, lb, ub, estimate := ApproximateStabiliser (G, A, U);
> #H, #B;
460800 230400
```

`StabiliserOfSpaces(Q)`

Determine the subgroup of $GL(d, F)$, for F a finite field, which stabilises the sequence Q of subspaces of the natural vector space. The function also returns generators for the largest unipotent subgroup of the stabiliser. For a description of this algorithm, see Schwingel [Sch00]; this implementation was prepared by Eamonn O'Brien.

Example H59E20

```
> V := VectorSpace(GF (3), 4);
> Spaces := [sub< V | [1,1,0,2]>, sub < V | [ 1, 0, 2, 0 ], [ 0, 1, 0, 0]>];
> S, P := StabiliserOfSpaces(Spaces);
> #S;
5184
> P;
[
    [1 1 0 0]
    [0 1 0 0]
    [0 1 1 0]
    [0 1 0 1],

    [2 0 2 0]
    [0 1 0 0]
    [1 0 0 0]
    [1 0 2 1],

    [1 0 1 2]
    [0 1 0 0]
    [0 0 2 2]
    [0 0 1 0]
]
```

Thus, the unipotent subgroup generated by P has order 3^3.

`IsUnipotent(G)`

If G is a p-subgroup of $GL(d, F)$, where F is a finite field of characteristic p, then return `true`, else return `false`.

`UnipotentStabiliser(G, U:` *parameters*`)`

Given a unipotent subgroup G of $GL(d, F)$, for F a finite field, U a subspace of the natural vector space, determine the stabiliser in G of U. The function returns the stabiliser in G of U, the *canonical* element C of the orbit of U under G, an element x of G such that $U^x = C$, and an SLP for x as an element of `WordGroup`(G). This function does *not* compute the orbit of U under G, but instead constructs the canonical element of the orbit. Hence it can be used to decide whether or not two subspaces belong to the same orbit. For a description of this algorithm, see [Sch00]; this implementation was prepared by Elliot Costi.

Example H59E21

```
> V := VectorSpace(GF (3), 4);
> G := sub< GL (4, 3) |
>     [ 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1 ],
>     [ 2, 0, 2, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 2, 1 ],
>     [ 1, 0, 1, 2, 0, 1, 0, 0, 0, 0, 2, 2, 0, 0, 1, 0 ] >;
> U := sub < V | [ 1, 2, 0, 1 ],[ 2, 2, 1, 0 ]>;
> S, C, x, w := UnipotentStabiliser(G, U);
> S;
MatrixGroup(4, GF(3))
Generators:
    [2 1 2 0]
    [0 1 0 0]
    [1 1 0 0]
    [1 1 2 1]
> #S;
3
> Index(G, S);
9
```

So the stabiliser of U has order 3 and U lies in an orbit of size 9. We print the canonical element of the orbit of U under G. The element x maps U to C and w evaluates to x.

```
> C;
Vector space of degree 4, dimension 2 over GF(3)
Echelonized basis:
(1 0 0 0)
(0 1 2 0)
>  U^x;
Vector space of degree 4, dimension 2 over GF(3)
Echelonized basis:
(1 0 0 0)
(0 1 2 0)
> W, phi := WordGroup (G);
>  phi (w);
[1 0 2 1]
[0 1 0 0]
[0 0 0 1]
[0 0 2 2]
```

59.10.3 Action on Orbits

`OrbitAction(G, T)`

Given a matrix group G with natural module M, and a set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T and construct the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the set Y. The function returns:

(a) The natural homomorphism $\phi : G \to L$;

(b) The induced group L;

(c) The kernel of the action (a subgroup of G).

`OrbitActionBounded(G, T, b)`

Given a matrix group G with natural module M, and a set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T. If the cardinality of Y does not exceed b, then construct the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the set Y. In this case the function returns:

(a) The boolean value `true`.

(b) The natural homomorphism $\phi : G \to L$;

(c) The induced group L;

(d) The kernel of the action (a subgroup of G). If the cardinality of Y exceeds b, simply return `false`. (The action of G on Y is not constructed in this case).

`OrbitImage(G, T)`

Given a matrix group G with natural module M, and a set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T and return the permutation group L giving the action of G on Y.

`OrbitImageBounded(G, T, b)`

Given a matrix group G with natural module M, and set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T. If the cardinality of Y does not exceed b, return `true` together with the permutation group L giving the action of G on Y. If the cardinality of Y does exceed b, the action is not constructed and the single value `false` is returned.

`OrbitKernel(G, T)`

Given a matrix group G with natural module M, and a set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T and return the the kernel of the action of G on Y.

`OrbitKernelBounded(G, T, b)`

Given a matrix group G with natural module M, and set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T. If the cardinality of Y does not exceed b, return the boolean value `true` together with the kernel of the action of G on Y. If the cardinality of Y does exceed b, the kernel is not constructed and the single value `false` is returned.

Example H59E22

We look for a small G-set for the group $J2A2$ (defined above) by examining eigenspaces of its generators. Having found a reasonably sized set, we then construct a permutation representation for G on this set.

```
> [ Factorization(CharacteristicPolynomial(G.i)) : i in [1..3] ];
[
    [
          <x^3 + w^5*x^2 + w^3*x + 2, 1>,
          <x^3 + w^7*x^2 + w*x + 2, 1>
    ],
    [
          <x + 2, 6>
    ],
    [
         <x + w^2, 3>,
         <x + w^6, 3>
    ]
]
> y := Eigenspace(G.2, -2);
> y;
Vector space of degree 6, dimension 3 over GF(3, 2)
Echelonized basis:
(1 0 0 1 2 1)
(0 1 0 2 1 2)
(0 0 1 1 2 1)
> #Orbit(G, y);
280
> P := OrbitImage(G, y);
> P;
Permutation group P of degree 280
> Order(P);
604800
> CompositionFactors(P);
    G
    |  J2
    1
```

Thus, our group has the simple group J_2 of Janko as a composition factor.

```
> Order(G);
```

```
1209600
```

Hence the kernel of this action has order 2.

59.10.4 Action on a Coset Space

`CosetAction(G, H)`

Given a subgroup H of the group G, construct the permutation representation of G given by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The natural homomorphism $f : G \to L$;

(b) The induced permutation group L;

(c) The kernel K of the action (a subgroup of G).

`CosetImage(G, H)`

Given a subgroup H of the group G, construct the image L of G given by the action of G on the set of (right) cosets of H in G. L is returned as a permutation group.

`CosetKernel(G, H)`

Given a subgroup H of the group G, construct the kernel of the action of G on the set of (right) cosets of H in G.

Example H59E23

We construct $G = \mathrm{SL}(3, 3)$, a subgroup H of G, and the permutation representation of G given by its action on the cosets of H.

```
> G := MatrixGroup< 3, GF(3) | [0,2,0, 1,1,0, 0,0,1], [0,1,0, 0,0,1, 1,0,0] >;
> Order(G);
5616
> H := sub< G | G.1^2, G.2 >;
> Order(H);
216
> P := CosetImage(G, H);
> P;
Permutation group P of degree 26
    (1, 2)(3, 4, 6, 5, 7, 9)(8, 11)(10, 13, 15, 20, 18, 17)
      (12, 16, 21, 14, 19, 24)(23, 26)
    (2, 3, 5)(4, 6, 8)(7, 10, 14)(9, 12, 17)(11, 15, 20)(13, 18, 23)
      (16, 22, 21)(19, 25, 24)
```

59.10.5 Action on the Natural G-Module

A set of functions is provided for working with the action of G on the natural G-module M, for a matrix group G defined over a finite field. Many of these functions are similar to those presented in the general module chapter.

`GModule(G)`

The natural $R[G]$-module M for the matrix group G.

`IsIrreducible(G)`

Given a matrix group G, return `true` iff G acts irreducibly on its natural module M. If G acts reducibly on M, a proper submodule S of M is also returned.

`SubmoduleAction(G, S)`

Given a matrix group G and a submodule S of the natural module M of G, return the action homomorphism f of G on S, together with the image of f.

`SubmoduleImage(G, S)`

Given a matrix group G and a submodule S of the natural module M of G, return the image of the action homomorphism of G on S.

`QuotientModuleAction(G, S)`

Given a matrix group G and a submodule S of the natural module M of G, return the quotient action homomorphism f of G on S, together with the image of f.

`QuotientModuleImage(G, S)`

Given a matrix group G and a submodule S of the natural module M of G, return the quotient image of the action homomorphism of G on S.

`IsAbsolutelyIrreducible(G)`

Given a matrix group G, return `true` if and only if G acts absolutely irreducibly on its natural module M. In addition, if G is absolutely irreducible, the function returns the (matrix algebra) generator of the endomorphism algebra E of M (which is always a field), and the dimension of E.

`AbsoluteRepresentation(G)`

Given an irreducible matrix group G, return the isomorphic reduced-degree absolute representation A of G, which is over the absolute field of the natural module M of G and is absolutely irreducible, together with the corresponding isomorphism.

`MinimalField(G)`

Given a matrix group G defined over a finite field K, return the minimal subfield of K over which G can be realised.

59.11 Normal and Subnormal Subgroups

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

59.11.1 Characteristic Subgroups and Subgroup Series

`Centre(G)`
`Center(G)`

Construct the centre of the group G.

`DerivedLength(G)`

The derived length of the matrix group G. If G is non-soluble, the function returns the number of terms in the series terminating with the soluble residual.

`DerivedSeries(G)`

The derived series of the group G. The series is returned as a sequence of subgroups.

`CommutatorSubgroup(G)`
`DerivedSubgroup(G)`
`DerivedGroup(G)`

The derived subgroup of the group G.

`#FittingSubgroup(G)`

The Fitting subgroup of the group G.

`LowerCentralSeries(G)`

The lower central series of the matrix group G. The series is returned as a sequence of subgroups.

`NilpotencyClass(G)`

The nilpotency class of the group G.

`H ^ G`
`NormalClosure(G, H)`

The normal closure of the subgroup H of group G.

`SolubleResidual(G)`
`SolvableResidual(G)`

The solvable residual of the group G.

`SubnormalSeries(G, H)`

Given a group G and a subnormal subgroup H of G, return a sequence of subgroups commencing with G and terminating with H, such that each subgroup is normal in the previous one. If H is not subnormal in G, the empty sequence is returned.

`UpperCentralSeries(G)`

The upper central series of the matrix group G. The series is returned as a sequence of subgroups. As the algorithm used requires the conjugacy classes of G, this function is much more restricted in its range of application than `DerivedSeries` and `LowerCentralSeries`.

Example H59E24

We demonstrate some of the series functions by applying them to a soluble subgroup of $\mathrm{GL}(3,5)$.

```
> G := MatrixGroup< 3, GF(5) | [0,1,0, 1,0,0, 0,0,1],
>                              [0,1,0, 0,0,1, 1,0,0],
>                              [2,0,0, 0,1,0, 0,0,1] >;
> Order(G);
384
> DerivedGroup(G);
MatrixGroup(3, GF(5, 1))
Generators:
[0 0 1]
[1 0 0]
[0 1 0]

[2 0 0]
[0 3 0]
[0 0 1]
> D := DerivedSeries(G);
> [ Order(d) : d in D ];
[ 384, 48, 16, 1 ]
> L := LowerCentralSeries(G);
> [ Order(l) : l in L ];
[ 384, 48 ]
> K := sub< G | [ 2,0,0, 0,3,0, 0,0,2 ] >;
> S := SubnormalSeries(G, K);
> [ Order(s) : s in S ];
[ 384, 16, 4 ]
```

59.11.2 The Soluble Radical and its Quotient

The functions in this section enable the user to construct the radical, its quotient and an elementary abelian series. They are currently restricted to matrix groups where a base and strong generating set can be constructed and the base ring is either a field or can be embedded into a field.

`Radical(G)`
`SolubleRadical(G)`
`SolvableRadical(G)`

Given a group G, return the maximal normal solvable subgroup of G. The algorithm is to compute the radical quotient map, and then compute its kernel. The algorithm used is described in Unger [Ung06b].

`RadicalQuotient(G)`

Given a group G, compute a permutation representation of the quotient G/R where R is the (solvable) radical of G. Both the permutation group Q isomorphic to G/R and a homomorphism $\phi : G \to Q$ are returned. The third return value is R, the radical of G and the kernel of the homomorphism. The algorithm used is described in Unger [Ung06b].

`ElementaryAbelianSeries(G: parameters)`

`LayerSizes` SeqEnum[RngIntElt] *Default :* []

An elementary abelian series is a chain of normal subgroups $R = N_1 > N_2 > ... > N_r = 1$ with the property that the quotient of each pair of successive terms in the series is elementary abelian and that there is no group $R < H < G$ such that H/R is elementary abelian and H normal in G. The top of the series R is called the solvable radical and is the maximal normal solvable subgroup of G.

The parameter `LayerSizes` controls possible refinement of the series. As an example, take `LayerSizes := [ 2, 5, 3, 4, 7, 3, 11, 2, 17, 1]`. When constructing an elementary abelian series for the group, attempt to split 2-layers of size gt 2^5, 3-layers of size gt 3^4, etc. The implied exponent for 13 is 2 and for all primes greater than 17 the exponent is 1. Setting `LayerSizes` to `[2, 1]` will attempt to split all layers, resulting in a portion of a chief series for G.

`ElementaryAbelianSeriesCanonical(G)`

Gives a similar result to using `ElementaryAbelianSeries`, except the series returned depends only on the isomorphism type of the solvable radical, and consists of characteristic subgroups of G. This function may be slower than `ElementaryAbelianSeries`.

59.11.3 Composition and Chief Factors

The functions in this section enable the user to find the composition factors of a matrix group. They are restricted to matrix groups where a base and strong generating set can be constructed. The chief series and factors functions are further restricted to groups where the base ring is either a field or can be embedded into a field.

`CompositionFactors(G)`

Given a matrix group G, return a sequence S of tuples that represent the composition factors of G, ordered according to some composition series of G. Each tuple is a triple of integers f, d, q that defines the isomorphism type of the corresponding composition factor. A triple $< f, d, q >$ describes a simple group as follows. The integer f defines the family to which the group belongs, and d and q are the parameters of the family. The length of the sequence S is the number of composition factors of G. The numbering of the simple group families is given in Tables 1 and 2 of the chapter on permutation groups on page 1590.

`ChiefFactors(G)`

Given a group G, return a sequence of the isomorphism types $< f, d, q, m >$ of the chief factors. An isomorphism type in a chief factor should be understood as the direct product of m copies of the simple group described by $< f, d, q >$ (see `CompositionFactors` above). For the algorithm, see Unger [Ung].

`ChiefSeries(G)`

Given a group G, return the chief series of G and a sequence of the corresponding isomorphism types $< f, d, q, m >$ of the chief factors. An isomorphism type in a chief factor should be understood as the direct product of m copies of the simple group described by $< f, d, q >$ (see `CompositionFactors` above). The series will be organised to contain the soluble radical of G, and, if G is insoluble, the socle of the quotient of G by the soluble radical.

Example H59E25

We get the chief factors of a group of degree 4 defined over the cyclotomic field of order 8.

```
> L<zeta_8> := CyclotomicField(8);
> w := -( - zeta_8^3 - zeta_8^2 + zeta_8);
> // Define sqrt(q)
> rt2 := -1/6*w^3 + 5/6*w;
> // Define sqrt(-1)
> ii := -1/6*w^3 - 1/6*w;
> f := rt2;
> t := f/2 + (f/2)*ii;
> GL4L := GeneralLinearGroup(4, L);
>
> A := GL4L ! [ 1/2, 1/2, 1/2, 1/2,
>             1/2,-1/2, 1/2,-1/2,
```

```
>             1/2, 1/2,-1/2,-1/2,
>             1/2,-1/2,-1/2, 1/2 ];
>
> B := GL4L ! [ 1/f,   0, 1/f,   0,
>               0, 1/f,   0, 1/f,
>             1/f,   0,-1/f,   0,
>               0, 1/f,   0,-1/f ];
>
> g4 := GL4L ! [  1, 0, 0, 0,
>                 0, 1, 0, 0,
>                 0, 0, 1, 0,
>                 0, 0, 0,-1 ];
>
> D1 := GL4L ! [ 1, 0, 0, 0,
>               0,ii, 0, 0,
>               0, 0, 1, 0,
>               0, 0, 0,ii ];
>
> D3 := GL4L ! [ t, 0, 0, 0,
>               0, t, 0, 0,
>               0, 0, t, 0,
>               0, 0, 0, t ];
>
> G3 := sub< GL4L | A, B, g4, D1, D3 >;
> Order(G3);
92160
> ChiefFactors(G3);
    G
    |  Cyclic(2)
    *
    |  Alternating(6)
    *
    |  Cyclic(2) (4 copies)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
```

59.12 Coset Tables and Transversals

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

`CosetTable(G, H)`

The (right) coset table for the group G over subgroup H relative to its defining generators.

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a matrix group G and a subgroup H of G, this function returns

(a) A set of elements T of G forming a right transversal for G over H; and

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and $g \in G$, ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

59.13 Presentations

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

59.13.1 Presentations

`FPGroup(G)`

Construct a presentation for the matrix group G on the set of defining generators and return the presentation in the form of a finitely presented group F that is isomorphic to G. The presentation is obtained by first computing the regular representation of G and then using the Todd-Coxeter Schreier algorithm to construct a presentation on the strong generators. In this situation the strong generators are identical to the defining generators.

A group homomorphism $\phi : F \to G$, defining G as a matrix representation of F, is also returned.

`FPGroupStrong(G)`

Construct a presentation for the matrix group G on a set of strong generators and return the presentation in the form of a finitely presented group F that is isomorphic to G. In MAGMA, the Todd-Coxeter Schreier algorithm is used to construct the presentation. If strong generators are not already known for G, they will be constructed. In the case in which strong generators are already known for G, the presentation will be on these strong generators.

The presentation will have the property that it contains presentations for all stabilizer subgroups defined by the BSGS.

The group homomorphism $f : F \to G$, defining G as a matrix representation of F, is also returned.

59.13.2 Matrices as Words

Consider a matrix group G defined on d generators. The *word group* of G is a free group W of rank d. Then we regard G as a homomorphic image of F with associated homomorphism $\phi : W \to G$. All operations involving words in the generators of G will be performed in W.

`WordGroup(G)`

Given a matrix group G defined on d generators, return (a) a free group W on d generators as an SLP-group, and (b) the homomorphism ϕ from W to G such that $W.i \to G.i$, for $i = 1, \ldots, d$. The group W associated with G by this function will be referred to as the *word group* for G.

`InverseWordMap(G)`

Given a matrix group G and its associated word group W with canonical homomorphism $\phi : W \to G$, construct the inverse mapping ρ. Thus, given a matrix g of G, $g@\rho$ returns an element in the preimage of g under ϕ. If the word group W does not already exist, it will be created.

59.14 Automorphism Groups

The automorphism group of a finite matrix group may be computed in Magma, subject to the same restrictions on the group as when computing maximal subgroups. (That is, all of the non-abelian composition factors of the group must appear in a certain database.) The methods used are those described in Cannon and Holt [CH03]. The existence of an isomorphism between a given matrix group and any other type of finite group (`GrpPerm` or `GrpPC`) may also be determined using similar methods.

`AutomorphismGroup(G: parameters)`

Given a finite matrix group G, construct the full automorphism group F of G. The function returns the full automorphism group of G as a group of mappings (i.e., as a group of type `GrpAuto`). The automorphism group F is also computed as a finitely presented group and can be accessed via the function `FPGroup(F)`. A function `PermutationRepresentation` is provided that when applied to F, attempts to construct a faithful permutation representation of reasonable degree. The algorithm described in Cannon and Holt [CH03] is used.

`SmallOuterAutGroup` RngIntElt *Default :* 20000

`SmallOuterAutGroup :=` t: Specify the strategy for the backtrack search when testing an automorphism for lifting to the next layer. If the outer automorphism group O at the previous level has order at most t, then the regular representation of O is used, otherwise the program tries to find a smaller degree permutation representation of O.

`Print` RngIntElt *Default :* 0

The level of verbose printing. The possible values are 0, 1, 2 or 3.

`PrintSearchCount` RngIntElt *Default :* 1000

`PrintSearchCount := s`: If `Print := 3`, then a message is printed at each s-th iteration during the backtrack search for lifting automorphisms.

Further information about the construction of the automorphism group and a description of machinery for computing with group automorphisms may be found in Chapter 67.

Example H59E26

We construct a 3-dimensional matrix group over $GF(4)$ and determine the order of its automorphism group.

```
> k<w> := GF(4);
> G := MatrixGroup< 3, k |
> [w^2, 0, 0, 0, w^2, 0, 0, 0, w^2],
> [w^2, 0, w^2, 0, w^2, w^2, 0, 0, w^2],
> [1, 0, 0, 1, 0, w, w^2, w^2, 0],
> [w, 0, 0, w^2, 1, w^2, w, w, 0],
> [w, 0, 0, 0, w, 0, 0, 0, w] >;
> G;
MatrixGroup(3, GF(2^2))
Generators:
    [w^2   0   0]
    [  0 w^2   0]
    [  0   0 w^2]

    [w^2   0 w^2]
    [  0 w^2 w^2]
    [  0   0 w^2]

    [  1   0   0]
    [  1   0   w]
    [w^2 w^2   0]

    [  w   0   0]
    [w^2   1 w^2]
    [  w   w   0]

    [  w   0   0]
    [  0   w   0]
    [  0   0   w]
> #G;
576
> A := AutomorphismGroup(G);
> #A;
3456
> OuterOrder(A);
72
> F := FPGroup(A);
```

```
> P := DegreeReduction(CosetImage(F, sub<F|>));
> P;
Permutation group P acting on a set of cardinality 48
```

Thus, we see that G has an automorphism group of order 3456 and the quotient group of A consisting of outer automorphisms, has order 72. The automorphism group may be realised as a permutation group of degree 48.

IsIsomorphic(G, H: *parameters*)

Test whether or not the two finite groups G and H are isomorphic as abstract groups. If so, both the result `true` and an isomorphism from G to H is returned. If not, the result `false` is returned. The algorithm described in Cannon and Holt [CH03] is used.

SmallOuterAutGroup	RngIntElt	*Default :* 20000

`SmallOuterAutGroup := t`: Specify the strategy for the backtrack search when testing an automorphism for lifting to the next layer. If the outer automorphism group O at the previous level has order at most t, then the regular representation of O is used, otherwise the program tries to find a smaller degree permutation representation of O.

Print	RngIntElt	*Default :* 0

The level of verbose printing. The possible values are 0, 1, 2 or 3.

PrintSearchCount	RngIntElt	*Default :* 1000

`PrintSearchCount := s`: If `Print := 3`, then a message is printed at each s-th iteration during the backtrack search for lifting automorphisms.

Example H59E27

We construct a 3-dimensional point group of order 8 and test it for isomorphism with the dihedral group of order 8 given as a permutation group.

```
> n := 4;
> N := 4*n;
> K<z> := CyclotomicField(N);
> zz := z^4;
> i := z^n;
> cos := (zz+ComplexConjugate(zz))/2;
> sin := (zz-ComplexConjugate(zz))/(2*i);
> GL := GeneralLinearGroup(3, K);
> G := sub< GL | [ cos, sin, 0,
>                 -sin, cos, 0,
>                    0,   0, 1 ],
>
>                [  -1,   0, 0,
>                    0,   1, 0,
>                    0,   0, 1 ] >;
```

```
>
>  #G;
8
> D8 := DihedralGroup(4);
> D8;
Permutation group G acting on a set of cardinality 4
Order = 8 = 2^3
    (1, 2, 3, 4)
    (1, 4)(2, 3)
> #D8;
8
> bool, iso := IsIsomorphic(G, D8);
> bool;
true
> iso;
Homomorphism of MatrixGroup(3, K) of order 2^3 into
GrpPerm: D8, Degree 4, Order 2^3 induced by
    [ 0  1  0]
    [-1  0  0]
    [ 0  0  1] |--> (1, 2, 3, 4)

    [-1  0  0]
    [ 0  1  0]
    [ 0  0  1] |--> (1, 3)
```

59.15 Representation Theory

A set of functions are provided for computing with the characters of a group. Full details of these functions may be found in Chapter 91. For convenience we include here two of the more useful character functions.

Also, functions are provided for computing with the modular representations of a group. Full details of these functions may be found in Chapter 89. For the reader's convenience we include here the functions which may be used to define a $K[G]$-module for a matrix group.

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

`LinearCharacters(G)`

A sequence containing the linear characters for the group G.

`CharacterTable(G: parameters)`

Construct the table of ordinary irreducible characters for the group G.

`Al` MonStgElt *Default :* `"Default"`

This parameter controls the algorithm used. The string `"DS"` forces use of the Dixon-Schneider algorithm. The string `"IR"` forces the use of Unger's induction/reduction algorithm [Ung06]. The `"Default"` algorithm is to use Dixon-Schneider for groups of order ≤ 5000 and Unger's algorithm for larger groups. This may change in future.

`DSSizeLimit` RngIntElt *Default :* 0

When the default algorithm is selected, a positive value n for `DSSizeLimit` means that before using Unger's algorithm, the full character space is split by some passes of Dixon-Schneider, restricted to using class matrices corresponding to conjugacy classes with size at most n.

`PermutationCharacter(G, H)`

Given a group G and a subgroup H of G, construct the ordinary character afforded by the representation of G given by its action on the coset space of the subgroup H.

`GModule(G)`

The natural $R[G]$-module for the matrix group G.

`GModule(G, A)`

Let A be a matrix ring defined over the ring R and let G be a finite group defined on m generators. Let M denote the underlying module of A. Suppose there is a one-to-one correspondence between the generators of G and the generators $[A_1, \ldots, A_m]$ of A. The function `GModule` creates the $R[G]$-module corresponding to an action of G on M defined by A, where the action of the i-th generator of G on M is given by A_i.

`GModule(G, Q)`

Let A be a matrix ring defined over the ring R and let G be a finite group defined on m generators. Let M denote the underlying module of A. Given a sequence Q of m elements of A, the function `GModule` creates the $R[G]$-module corresponding to an action of G on M defined by Q, where the action of the i-th generator of G on M is given by $Q[i]$.

`GModule(G, A, B)`

Let A and B be normal subgroups of G such that B is contained in A. Further, assume that A/B is elementary abelian of order p^n, p a prime. Let K denote the field of p elements. This function constructs a $K[G]$-module corresponding to the action of the group G on the elementary abelian section A/B of G. The map from A to the $K[G]$-module's underlying vector space is also returned.

`PermutationModule(G, H, R)`

The permutation module for the matrix group G over the ring R defined by its action on the cosets of the subgroup H.

`ChangeOfBasisMatrix(G, S)`

Given a matrix group G and a submodule S of its natural module, return an invertible matrix with topmost rows a basis for S. Conjugating by the inverse of this matrix puts the generators of G into a block form that exhibits their action on S and the quotient module.

Example H59E28

We use the module machinery to refine an elementary abelian normal subgroup by finding a normal subgroup contained in it.

```
> G := MatrixGroup<4, IntegerRing(4) |
>  [ 3, 3, 1, 3, 0, 2, 2, 3, 3, 0, 1, 3, 3, 2, 2, 1 ],
>  [ 2, 2, 3, 3, 0, 3, 1, 1, 3, 0, 1, 1, 2, 0, 1, 2 ] >;
> #G;
660602880
> H := pCore(G, 2);
> FactoredOrder(H);
[ <2, 15> ]
> IsElementaryAbelian(H);
true
> M, f := GModule(G, H, sub<H|>);
> SM := Submodules(M);
> #SM;
3
```

One of these submodules is 0, one is all `M`, we are interested in the one in the middle. Note that the result returned by `Submodules` is sorted by dimension.

```
> N := SM[2] @@ f;
> N;
MatrixGroup(4, IntegerRing(4))
Generators:
  [3 0 0 0]
  [0 3 0 0]
  [0 0 3 0]
  [0 0 0 3]
```

We have found `N`, a normal subgroup of `G`, contained in the 2-core, with order 2.

59.16 Base and Strong Generating Set

59.16.1 Introduction

Computing structural information for a matrix group G requires, in most cases, a representation of the set of elements of G. MAGMA represents this set by means of a *base and strong generating set, or BSGS* for G. Suppose the group G has the natural module M. A *base* B for G is a sequence of distinct elements and submodules of M with the property that the identity is the only element of G that fixes B pointwise. A base B of length n determines a sequence of subgroups $G^{(i)}$, $1 \leq i \leq n+1$, where $G^{(i)}$ is the stabilizer of the first $i-1$ points of B. Given a base B for G, a subset S of G is said to be a *strong generating set* for G if $G^{(i)} = \langle S \cap G^{(i)} \rangle$, for $i = 1, \ldots, n$.

Unlike permutation groups, however, the orbits of the i-th base point under the stabilizer $G^{(i)}$ are not bounded by the degree, but rather, by (where the base point is a 1-dimensional subspace) $(q^n - 1)/(q - 1)$ where q is the cardinality of the coefficient field and n is the degree of G. Clearly, it is essential to find small orbits if one is to compute with matrix groups in this manner. Unfortunately, there are no methods which are guaranteed to find short orbits. There are, however, some heuristics developed by Scott Murray and Eamonn O'Brien which often find good base points. These heuristics are used in MAGMA if the most likely standard base point would generate an orbit longer than 10000 (this bound may be changed).

59.16.2 Controlling Selection of a Base

Given the difficulties in automatically finding a good base for a matrix group, it is possible to apply the Murray-O'Brien base point selection procedure and preset a suitable base manually.

`GoodBasePoints(G: parameters)`

Apply the Murray–O'Brien base point selection procedure and return a sequence of vectors or subspaces according to the parameters. The procedure computes and sorts a collection of eigenspaces $[V_1, \ldots, V_m]$ for a generating set for the matrix group G. The default action is then to return $[V_1.1, \ldots, V_m.1, V_1.2, \ldots]$ where each new vector is only added if it is not in the span of the preceding vectors.

`Slots` RNGINTELT *Default :* 10

Expand the number of generators to work with to `Slots` matrices by adding random words in the generators of G.

`NoCycle` RNGINTELT *Default :* `false`

If `NoCycle := true`, instead of cycling through the eigenspaces, return the sequence $[V_1.1, \ldots, V_1.(\dim V_1), V_2.1, \ldots]$, with the addition of each vector subject to the same condition above.

`Eigenspaces` RNGINTELT *Default :* `false`

If `Eigenspaces := true`, then return the subsequence of the eigenspaces where all the eigenspaces have dimension $d \leq 10$. If there are no such eigenspaces, all the eigenspaces are returned.

```
AssertAttribute(G, "Base", B)
```

Set the base of the matrix group G to be $[B[1], \ldots, B[n]]$ where the tuple B has n components. An error will be reported if the matrix group G already has a base set.

```
HasAttribute(G, "Base")
```

Return whether the matrix group G has a base set, and if so, the base.

```
AssertAttribute(GrpMat, "FirstBasicOrbitBound", n)
```

Set the limit for the size of the first basic orbit to be n. If n is non-zero and the orbit of the first base point (a 1-dimensional subspace generated by a standard basis vector) has length exceeding n, then the Murray-O'Brien base point selection procedure is used to find a point more likely to have a short orbit. This assertion will affect all matrix groups. If $n = 1$ then use of the Murray-O'Brien procedure is guaranteed.

```
HasAttribute(GrpMat, "FirstBasicOrbitBound")
```

Get the limit for the size of the first basic orbit. This will always return `true` and the limit.

59.16.3 Construction of a Base and Strong Generating Set

The functions described below give user control of the construction of a base and strong generating set (BSGS) of a finite matrix group.

Many functions described in this chapter require a group to have a BSGS. In case the given group does not have a BSGS, then one will be constructed using the default algorithm, which is equivalent to using the `BSGS` procedure described below.

It should be noted that if the user constructs a BSGS for a group G using the `RandomSchreier` procedure, then other functions that require a BSGS will assume that the random BSGS is a complete BSGS. If this is not the case then results will be unpredictable.

```
BSGS(G)
```

```
BSGS(G, str)
```

The general procedure for constructing a base and strong generating set for the matrix group G. This version uses the default algorithm choices. Currently this is as follows: if the order of the group is known to the program then a BSGS is constructed using the random Schreier algorithm, if not then the Sims-Todd-Coxeter-Schreier procedure is used. If `str` is the name of a sporadic group, we assume that G is a representation for this group and choose base points specific to this group. This should ensure better performance. Information on the progress of these algorithms may be obtained by setting the verbose flags `RandomSchreier` and `STCS` true.

`RandomSchreier(G: parameters)`

`RandomSchreier(G, str : parameters)`

`Run` RngIntElt *Default :* 40

Construct a probable base and strong generating set for the group G. The strong generators are constructed from a set of randomly chosen elements of G. The expectation is that if sufficiently many random elements are taken then, upon termination, the algorithm will have produced a BSGS for G. If the attribute Order is defined for G, the random Schreier will continue until a BSGS defining a group of the indicated order is obtained. In such circumstances this method is the fastest method of constructing a base and strong generating set for G. This is particularly so for groups of large degree. If nothing is known about G, the random Schreier algorithm provides a cheap way of obtaining lower bounds on the group's order. This procedure has one associated parameter `Run`, which takes a positive integer value. If the value of `Run` is n, then the algorithm terminates after n consecutive random elements are found to lie in the set defined by the current BSGS (default 40). This will happen even if the Order attribute is defined for G. It should be emphasized that unpredictable results may arise if the user uses the base and strong generators produced by this algorithm, when, in fact, it does not constitute a complete BSGS for G. The `Verify` procedure, described below, may be used to check the completeness of the BSGS constructed by this function.

If `str` is the name of a sporadic group, we assume that G is a representation for this group and choose base points specific to this group. This should ensure better performance.

Information on the progress of this algorithm may be obtained by setting the verbose flag `RandomSchreier` to true.

`ToddCoxeterSchreier(G)`

Construct a BSGS for the matrix group G using the Sims-Todd-Coxeter-Schreier algorithm. Information on the progress of this algorithm may be obtained by setting the verbose flag `STCS` to true.

`Verify(G)`

Given a matrix group G for which a possible BSGS is stored, verify the correctness of the BSGS. If it is not complete, proceed to complete it. The Sims-Todd-Coxeter-Schreier method is used.

If G has no BSGS stored, then use of `Verify` is equivalent to using the `BSGS` procedure described above.

Information on the progress of these algorithms may be obtained by setting the verbose flags `RandomSchreier` and `STCS` true.

59.16.4 Defining Values for Attributes

```
AssertAttribute(G, "Order", n)
```
```
AssertAttribute(G, "Order", Q)
```

Define the order of the matrix group G to be the integer n (factored integer Q).

```
AssertAttribute(G, "IsVerified", b)
```

If the boolean variable b is `true`, the existing pseudo strong generators for the matrix group G (possibly created by `RandomSchreier`) are to be taken as correct.

```
HasAttribute(G, "Order")
```
```
HasAttribute(G, "FactoredOrder")
```

Returns `true` iff the order of the group G is known. In that case, the order is also returned as the second value of the function.

```
HasAttribute(G, "IsVerified")
```

Returns `true` iff the matrix group G has a verified set of strong generators.

59.16.5 Accessing the Base and Strong Generating Set

```
Base(G)
```

A base for the matrix group G. The base is returned as a sequence of points of Ω. If a base is not known, one will be constructed.

```
BasePoint(G, i)
```

The i-th base point for the matrix group G. A base and strong generating set must be known for G.

```
BasicOrbit(G, i)
```

The basic orbit at level i as defined by the current base for the matrix group G. This function assumes that a BSGS is known for G.

```
BasicOrbitLength(G, i)
```

The length of the basic orbit at level i as defined by the current base for the matrix group G. This function assumes that a BSGS is known for G.

```
BasicOrbitLengths(G)
```

The lengths of the basic orbits as defined by the current base for the matrix group G. This function assumes that a BSGS is known for G. The lengths are returned as a sequence of integers.

`BasicStabilizer(G, i)`

`BasicStabiliser(G, i)`

Given a matrix group G for which a base and strong generating set are known, and an integer i, where $1 \leq i \leq k$ with k the length of the base, return the subgroup of G which fixes the first $i-1$ points of the base.

`BasicStabilizerChain(G)`

`BasicStabiliserChain(G)`

Given a matrix group G, return the stabilizer chain defined by the base as a sequence of subgroups of G. If a BSGS is not already known for G, it will be created.

`NumberOfStrongGenerators(G)`

`Nsgens(G)`

The number of elements in the current strong generating set for the matrix group G.

`StrongGenerators(G)`

A set of strong generators for the matrix group G. If they are not currently available, they will be computed.

59.17 Soluble Matrix Groups

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

59.17.1 Conversion to a PC-Group

`PolycyclicGenerators(G)`

Construct a polycyclic generating sequence for the soluble group G.

`PCGroup(G)`

Given a soluble group G, construct a group S in category `GrpPC`, isomorphic to G. In addition to returning S, the function returns an isomorphism $\phi : G \to S$.

59.17.2 Soluble Group Functions

`pCentralSeries(G, p)`

Given a soluble group G, and a prime p dividing $|G|$, return the lower p-central series for G. The series is returned as a sequence of subgroups.

59.17.3 p-group Functions

`IsSpecial(G)`

Given a p-group G, return `true` if G is special, `false` otherwise.

`IsExtraSpecial(G)`

Given a p-group G, return `true` if G is extraspecial, `false` otherwise.

`FrattiniSubgroup(G)`

Given a p-group G, return the Frattini subgroup.

`JenningsSeries(G)`

Given a p-group G, return the Jennings series for G. The series is returned as a sequence of subgroups.

59.17.4 Abelian Group Functions

`AbelianInvariants(G)`

`Invariants(G)`

Given an abelian group G, return a sequence Q containing the types of each p-primary component of G.

59.18 Bibliography

[**But76**] Gregory Butler. The Schreier Algorithm for Matrix Groups. In *Proceedings of SYMSAC '76*, pages 167–170, 1976.

[**CCH01**] J.J. Cannon, B. Cox, and D.F. Holt. Computing the subgroups of a permutation group. *J. Symb. Comp.*, 31:149–161, 2001.

[**CH03**] J.J. Cannon and D.F. Holt. Automorphism group computation and isomorphism testing in finite groups. *J. Symbolic Comp.*, 35(3):241–267, 2003.

[**CLG97**] Frank Celler and Charles R. Leedham-Green. Calculating the Order of an Invertible Matrix. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 55–60. AMS, 1997.

[**CLGM+95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

[**ELGO02**] Bettina Eick, C.R. Leedham-Green, and E.A. O'Brien. Constructing automorphism groups of a p-groups. *Comm. Algebra*, 30:2271–2295, 2002.

[**KP02**] J. Kuzmanovich and A. Pavlichenkov. Finite groups of matrices whose entries are integers. *Amer. Math. Monthly*, 109(2):173–186, 2002.

[**LGO02**] C.R. Leedham-Green and E.A. O'Brien. Recognising tensor-induced matrix groups. *J. Algebra*, 253:14–30, 2002.

[**LGPS91**] C.R. Leedham-Green, C.E. Praeger, and L.H. Soicher. Computing with group homomorphisms. *J. Symbolic Comp.*, 12(4/5):527–532, 1991.

[**MN89**] M. Mecky and J. Neubüser. Some remarks on the computation of conjugacy classes of soluble groups. *Bull. Austral, Math. Soc.*, 40(2):281–292, 1989.

[**MO95**] Scott H. Murray and E. A. O'Brien. Selecting base points for the Schreier-Sims algorithm for matrix groups. *J. Symbolic Comp.*, 6:577–584, 1995.

[**O'B90**] E.A. O'Brien. The p-group generation algorithm. *J. Symbolic Comput.*, 9:677–698, 1990.

[**RD04**] Colva M. Roney-Dougal. Conjugacy of subgroups of the general linear group. *Experiment. Math.*, 13:151–163, 2004.

[**Sch00**] Ruth Schwingel. *Two matrix group algorithms with applications to computing the automorphism group of a finite p-group.* PhD thesis, Queen Mary and Westfield College, University of London, 2000.

[**Ung**] W.R. Unger. Computing chief series of a large permutation group. In preparation.

[**Ung06a**] W.R. Unger. Computing the character table of a finite group. *J. Symbolic Comp.*, 41(8):847–862, 2006.

[**Ung06b**] W.R. Unger. Computing the solvable radical of a permutation group. *J. Algebra*, 300(1):305–315, 2006.

60 MATRIX GROUPS OVER FINITE FIELDS

Chapter 60
MATRIX GROUPS OVER FINITE FIELDS

60.1 Introduction

If a matrix group G is defined over a finite field then, provided that the group is not too large, we can construct a BSGS-representation for G and consequently apply the standard algorithms for group structure as described in Chapter 59. However, there are many examples of groups having moderately small dimension where we cannot find a BSGS-representation.

In this chapter we describe techniques for computing with matrix groups that do not assume that a BSGS-representation is available. Thus, the techniques described here apply to matrix groups possibly having much larger order or much larger dimension than those that can be handled with the techniques of Chapter 59.

The `CompositionTree` package introduced in Section 60.6, which includes the collection of LMG (large matrix group) functions described in Section 60.7, provides a framework for such investigations. The package was prepared by Henrik Bäärnhielm, Derek Holt, C.R. Leedham-Green and E.A. O'Brien, and includes code prepared by Peter Brooksbank, Elliot Costi, Heiko Dietrich, Alice Niemeyer, and Csaba Schneider.

For recent surveys of work in this area, we refer the reader to [O'B06, O'B11].

The techniques described in this chapter fall roughly into two categories.

(a) Functions based on Aschbacher's theorem classifying maximal subgroups of the general linear group. The main thrust of this work is to devise a framework for computing arbitrary structural information for a matrix group without the use of a BSGS-representation.

(b) Functions which employ Monte Carlo and Las Vegas algorithms to determine some property of the group.

60.2 Finding Elements with Prescribed Properties

`RandomElementOfOrder(G, n : parameters)`

`Central`	BoolElt	*Default :* `false`
`Proof`	BoolElt	*Default :* `true`
`Randomiser`	GrpRandProc	*Default :*
`MaxTries`	RngIntElt	*Default :* 100

Given a finite matrix group G, this intrinsic attempts to locate an element x of order n in G by random search. If such an element is found, then the return values are the boolean value `true`, the element x, and an SLP for this element.

If `Central` is `true`, then an element is sought which has order n modulo the centre of G. If `Proof` is `false`, then the element returned may have order a multiple of n. In either case, the final return value indicates whether the element returned is known to have the precise order. The parameter `MaxTries` specifies the maximum number of random elements that are chosen. The parameter `Randomiser` specifies the random process that is to be used to construct the element and the SLP returned for the element is in the word group associated with this process. The default value of `Randomiser` is the process `RandomProcessWithWords(G)`.

`RandomElementOfNormalClosure(G, N)`

Given a group G and a subgroup N of G, this intrinsic returns a random element of the normal closure of N in G. Note that G may be a permutation or matrix group. The algorithm is due to Leedham-Green and O'Brien [LGO02].

`InvolutionClassicalGroupEven(G : parameters)`

`SmallCorank`	BoolElt	*Default :* `false`
`Case`	MonStgElt	*Default :* "*unknown*"

Let G be a quasisimple classical group in its natural representation and in even characteristic. If G is of type Ω^+ or Ω^- then it must have even degree at least 4 and be defined over a field with at least 4 elements. The corank of an involution I is the rank of $I - \mathtt{Identity}(G)$. This function returns an involution I of corank in $[d/4, \ldots, d/2]$, the SLP for I in `WordGroup`(G), and the corank of the involution. The parameter `Case` should be one of `"SL"`, `"Sp"`, `"SU"`, `"Omega-"`, or `"Omega+"`. If `SmallCorank` is `true`, then accept involution of small corank. The algorithm used to construct the involution is described in [DLLGO13]; it was implemented by Heiko Dietrich.

60.3 Monte Carlo Algorithms for Subgroups

`CentraliserOfInvolution(G, g : parameters)`

`Central`	BoolElt	*Default :* `false`
`NumberRandom`	RngIntElt	*Default :* 100
`CompletionCheck`	UserProgram	*Default :*

Given an involution g in G, this function returns the centraliser C of g in G using an algorithm of John Bray [Bra00]. Since it is Monte Carlo, it may return only a subgroup of the centraliser. If `Central` is `true`, the projective centraliser of g will be constructed: its elements commute with g modulo the centre of G.

The optional argument `CompletionCheck` is a function which can be used to determine when we have constructed the centraliser. It takes the following arguments: the parent group G; the proposed centraliser C; the involution g. By default, the algorithm completes when 20 generators have been found for the centraliser or when `NumberRandom` elements have been considered.

`CentraliserOfInvolution(G, g, w : parameters)`

`Randomiser`	GrpRandProc	*Default :*
`Central`	BoolElt	*Default :* `false`
`NumberRandom`	RngIntElt	*Default :* 100
`CompletionCheck`	UserProgram	*Default :*

Given an involution g in a matrix group G together with a SLP w corresponding to g, this function returns the centraliser C of g in G and SLPs for the generators of C. The algorithm used is due to John Bray [Bra00]. Since it is Monte Carlo, it may return a proper subgroup of the centraliser. If `Central` is `true`, the projective centraliser of g is constructed: its elements commute with g modulo the centre of G.

The parameter `Randomiser` specifies the random process that is to be used to construct the centraliser. By default `Randomiser` is the value returned by `RandomProcessWithWords (G)`. The SLP for g must lie in the word group of this process. The optional argument `CompletionCheck` is a function which can be used to determine when we have constructed the centraliser. It takes four arguments: the parent group G; the proposed centraliser C; the involution g; and the list of the SLPs for the generators of C. By default, the algorithm completes when 20 generators have been found for the centraliser or when `NumberRandom` elements have been considered.

`AreInvolutionsConjugate(G, x, wx, y, wy : parameters)`

`Randomiser`	GrpRandProc	*Default :*
`MaxTries`	RngIntElt	*Default :* 100

This Monte Carlo algorithm attempts to construct an element c of the group G which conjugates the involution x to the involution y. The corresponding SLPs for x and y are wx and wy respectively. If such an element c is found, then three values are returned: `true`, c and the SLP for c. Otherwise, the boolean value `false` is returned. At most `MaxTries` random elements are considered.

The parameter `Randomiser` specifies the random process to be used. By default `Randomiser` is the value returned by `RandomProcessWithWords (G)`. The SLPs for x and y must lie in the word group of this process and the SLP for c will also lie in this word group.

`NormalClosureMonteCarlo(G, H )`

`NormalClosureMonteCarlo(G, H : parameters)`

`slpsH`	[]	*Default :* []
`ErrorProb`	FltRatElt	*Default :* 9/10
`SubgroupChainLength`	RngIntElt	*Default :* `Degree(H)`

This Monte Carlo algorithm constructs the normal closure N of H in G. If SLPs of the generators of H in the generators of G are supplied via the parameter

slpsH, then the function also returns SLPs for the generators of N. The parameter SubgroupChainLength is used to specify an upper bound on the length of any subgroup chain in H. The probability that N is a proper subgroup of the normal closure is bounded above by ErrorProb, assuming that SubgroupChainLength is correctly set.

DerivedGroupMonteCarlo(G : *parameters*)

Randomiser	GrpRandProc	*Default :*
NumberGenerators	RngIntElt	*Default :* 10
MaxGenerators	RngIntElt	*Default :* 100

Given a matrix group G defined over a finite field, this intrinsic returns the derived group of G, and a list of SLPs of its generators in the generators of G. The SLPs are elements of the word group of the random process. The algorithm is Monte Carlo and may return a proper subgroup of the derived group. The parameter Randomiser specifies the random process to be used. By default Randomiser is the value returned by RandomProcessWithWords (G). At least NumberGenerators and at most MaxGenerators will be constructed for the derived group.

IsProbablyPerfect(G : *parameters*)

NumberRandom	RngIntElt	*Default :* 100

This intrinsic attempts to prove that a matrix or permutation group G is perfect by establishing that its generators are in G'. Since it is Monte-Carlo, there is a small probability of error. If the function returns true, then G is perfect; if it returns false, then G might still be perfect. Each call considers NumberRandom random elements.

The algorithm is due to Leedham-Green and O'Brien [LGO02] and it uses NormalSubgroupRandomElement.

Example H60E1

We illustrate IsProbablyPerfect with a subgroup of $GU(4, 9)$.

```
> G := GU(4, 9);
> N := sub<G | (G.1, G.2)>;
```

The generators of N have been chosen to be a normal generating set for the derived group of G.

```
> IsProbablyPerfect(N);
false
> x := NormalSubgroupRandomElement(G, N);
> x;
[$.1^68 $.1^34 $.1^26 $.1^55]
[$.1^23 $.1^78 $.1^16 $.1^72]
[$.1^42  $.1^2 $.1^24      2]
[$.1^11 $.1^66 $.1^13 $.1^29]
>  L := sub< G | N, x>;
```

```
> IsProbablyPerfect(L);
true
```

We now consider $SO(7,5)$ and $\Omega(7,5)$.

```
> G := SO(7, 5);
> IsProbablyPerfect(G);
false
> G := Omega(7, 5);
> IsProbablyPerfect(G);
true
```

60.4 Aschbacher Reduction

60.4.1 Introduction

An on-going international research project seeks to develop algorithms to explore the structure of groups having either large order or large degree. The approach relies on the following theorem of Aschbacher [Asc84]:

A matrix group G acting on the finite dimensional $K[G]$-module V over a finite field K satisfies at least one of the following conditions (which we have simplified slightly for brevity):

(i) G acts reducibly on V;

(ii) G acts semilinearly over an extension field of K;

(iii) G acts imprimitively on V;

(iv) G preserves a nontrivial tensor-product decomposition of V;

(v) G has a normal subgroup N, acting absolutely irreducibly on V, which is an extraspecial p-group or 2-group of symplectic type;

(vi) G preserves a tensor-induced decomposition of V;

(vii) G acts (modulo scalars) linearly over a proper subfield of K;

(viii) G contains a classical group in its natural action over K;

(ix) G is almost simple modulo scalars.

The philosophy underpinning the research program is to attempt to decide that G lies in at least one of the above categories, and to calculate the associated isomorphism or decomposition explicitly.

Groups in Category (i) can be recognised easily by means of the Meataxe functions described in the chapter on R-modules.

Groups which act irreducibly but not absolutely irreducibly on V fall theoretically into Category (ii), and furthermore act linearly over an extension field of K. In fact, absolute irreducibility can be tested using the built-in MAGMA functions and, by redefining their

field to be an extension field L of K and reducing, they can be rewritten as absolutely irreducible groups of smaller dimension, but over L instead of K. We can therefore concentrate on absolutely irreducible matrix groups.

The `CompositionTree` package currently includes functions which seek to decide membership in all categories.

60.4.2 Primitivity

Let G be a subgroup of $\mathrm{GL}(d, q)$ and assume that G acts irreducibly on the underlying vector space V. Then G acts *imprimitively* on V if there is a non-trivial direct sum decomposition

$$V = V_1 \oplus V_2 \oplus \ldots \oplus V_r$$

where $V_1, \ldots, V_r$ are permuted by G. In such a case, each block V_i has the same dimension or size, and we have the *block system* $\{V_1, \ldots, V_r\}$. If no such system exists, then G is *primitive*.

Theoretical details of the algorithm used may be found in Holt, Leedham-Green, O'Brien, and Rees [HLGOR96b].

`SetVerbose ("Smash", 1)` will provide information on the progress of the algorithm.

`IsPrimitive(G:` *parameters*`)`

`BlockSizes` [RngIntElt] *Default :* []

Given a matrix group G defined over a finite field, this intrinsic returns `true` if G is primitive, `false` if G is not primitive, or `"unknown"` if no decision can be reached.

If `BlockSizes` is supplied, then the search is restricted to systems of imprimitivity whose block sizes are given in the sequence `BlockSizes` only. Otherwise all valid sizes will be considered.

`ImprimitiveBasis(G)`

Given a matrix group G defined over a finite field which is imprimitive, this intrinsic returns the change-of-basis matrix which exhibits the block structure for G.

`Blocks(G)`

Given a matrix group G defined over a finite field which is imprimitive, this intrinsic returns the blocks of imprimitivity of G.

`BlocksImage(G)`

Given a matrix group G defined over a finite field which is imprimitive, this intrinsic returns the group induced by the action of G on the system of imprimitivity.

`ImprimitiveAction(G, g)`

Given a matrix group G defined over a finite field which is imprimitive and an element g of G, this intrinsic returns action of g on blocks of imprimitivity as a permutation.

Example H60E2

We construct an imprimitive group by taking the wreath product of $GL(4, 7)$ with S_3.

```
> MG := GL (4, 7);
> PG := Sym (3);
> G := WreathProduct (MG, PG);
>
> IsPrimitive (G);
false
```

We investigate the block system for G.

```
> B := Blocks (G);
> B;
> #B;
4
> B[1];
Vector space of degree 12, dimension 4 over GF(7)
Generators:
(0 0 0 0 1 0 0 0 0 0 0 0)
(0 0 0 0 0 1 0 0 0 0 0 0)
(0 0 0 0 0 0 1 0 0 0 0 0)
(0 0 0 0 0 0 0 1 0 0 0 0)
Echelonized basis:
(0 0 0 0 1 0 0 0 0 0 0 0)
(0 0 0 0 0 1 0 0 0 0 0 0)
(0 0 0 0 0 0 1 0 0 0 0 0)
(0 0 0 0 0 0 0 1 0 0 0 0)
```

Now we obtain a permutation representation of G in its action on the blocks.

```
> P := BlocksImage (G);
> P;
Permutation group P acting on a set of cardinality 3
    (1, 2, 3)
    (2, 3)
> g := G.4 * G.3;
> ImprimitiveAction (G, g);
(1, 2)
```

60.4.3 Semilinearity

Let G be a subgroup of $\mathrm{GL}(d, q)$ and assume that G acts absolutely irreducibly on the underlying vector space V. Assume that a normal subgroup N of G embeds in $\mathrm{GL}(d/e, q^e)$, for $e > 1$, and a $d \times d$ matrix C acts as multiplication by a scalar λ (a field generator of $\mathbf{F}_{q^e}$) for that embedding.

We say that G acts as a *semilinear* group of automorphisms on the d/e-dimensional space if and only if, for each generator g of G, there is an integer $i = i(g)$ such that $Cg = gC^i$, that is, g corresponds to the field automorphism $\lambda \to \lambda^i$. If so, we have a map from G to the (cyclic) group $\mathrm{Aut}(GF(q^e))$, and C centralises the kernel of this map, which thus lies in $\mathrm{GL}(d, q^e)$.

Theoretical details of the algorithm used may be found in Holt, Leedham-Green, O'Brien and Rees [HLGOR96a].

`SetVerbose ("SemiLinear", 1)` will provide information on the progress of the algorithm.

`IsSemiLinear(G)`

Given a matrix group G defined over a finite field, this intrinsic returns `true` if G is semilinear, `false` if G is not semilinear, or `"unknown"` if no decision can be reached.

`DegreeOfFieldExtension(G)`

Let G be a subgroup of $K = \mathrm{GL}(d, q)$. The intrinsic returns the degree e of the extension field of F_q over which G is semilinear.

`CentralisingMatrix(G)`

Let G be a semilinear subgroup of $K = \mathrm{GL}(d, q)$. The intrinsic returns the matrix C which centralises the normal subgroup of G which acts linearly over the extension field of F_q.

`FrobeniusAutomorphisms(G)`

Let G be a semilinear subgroup of $K = \mathrm{GL}(d, q)$ and let C be the corresponding centralising matrix. The intrinsic returns a sequence S of positive integers, one for each generator g_i of G. The element $S[i]$ is the least positive integer such that $g_i^{-1} C g_i = C^{S[i]}$.

`WriteOverLargerField(G)`

Let G be a semilinear subgroup of $\mathrm{GL}(d, q)$ with extension degree e. This intrinsic returns:

(i) The normal subgroup N of the matrix group G which is the kernel of the map from G to C_e; this subgroup acts linearly over the extension field of K and is precisely the centraliser of C in G.

(ii) A cyclic group E of order e which is isomorphic to G/N.

(iii) A sequence of images of the generators of G in E.

Example H60E3

We analyse a semilinear group.

```
> P := GL(6,3);
> g1 := P![0,1,0,0,0,0,-1,0,0,0,0,0,
>          0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1];
> g2 := P![-1,0,0,0,1,0,0,-1,0,0,0,1,
>          -1,0,0,0,0,0,0,-1,0,0,0,0,0,0,-1,0,0,0,0,0,0,-1,0,0];
> g3 := P![1,0,0,0,0,0,0,-1,0,0,0,0,
>          0,0,1,0,0,0,0,0,0,-1,0,0,0,0,0,0,1,0,0,0,0,0,0,-1];
> G := sub <P | g1, g2, g3 >;
>
> IsSemiLinear (G);
true
> DegreeOfFieldExtension (G);
2
> CentralisingMatrix (G);
[2 2 0 0 0 0]
[1 2 0 0 0 0]
[0 0 2 2 0 0]
[0 0 1 2 0 0]
[0 0 0 0 2 2]
[0 0 0 0 1 2]
> FrobeniusAutomorphisms (G);
[ 1, 1, 3 ]
> K, E, phi := WriteOverLargerField (G);
```

The group K is the kernel of the homomorphism from G into E.

```
> K.1;
[0 1 0 0 0 0]
[2 0 0 0 0 0]
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
```

The return value E is the cyclic group of order e while *phi* gives the sequence of images of $G.i$ in E.

```
> E;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*E.1 = 0
>
> phi;
[ 0, 0, E.1 ]
```

60.4.4 Tensor Products

Let G be a subgroup of $\mathrm{GL}(d, K)$, where $K = GF(q)$, and let V be the natural $K[G]$-module. We say that G preserves a tensor decomposition of V as $U \otimes W$ if there is an isomorphism of V onto $U \otimes W$ such that the induced image of G in $\mathrm{GL}(U \otimes W)$ lies in $\mathrm{GL}(U) \circ \mathrm{GL}(W)$.

Theoretical details of the algorithm used may be found in Leedham-Green and O'Brien [LGO97b, LGO97a].

The verbose flag `SetVerbose ("Tensor", 1)` will provide information on the progress of the algorithm.

`IsTensor(G: parameters)`

`Factors` [SeqEnum] *Default :* []

Given a matrix group G defined over a finite field, this intrinsic returns `true` if G preserves a non-trivial tensor decomposition, `false` if G is does not preserve a tensor decomposition, or `"unknown"` if no decision can be reached.

A sequence of valid dimensions for potential factors may be supplied using the parameter `Factors`. Then for each element $[u, w]$ of the sequence `Factors`, the algorithm will search for decompositions of V as $U \otimes W$, where U must have dimension u and W must have dimension w only. If this parameter is not set, then all valid factorisations will be considered.

`TensorBasis(G)`

Given a matrix group G defined over a finite field that admits a tensor decomposion, this intrinsic returns the change-of-basis matrix which exhibits the tensor decomposition of G.

`TensorFactors(G)`

Given a matrix group G defined over a finite field that admits a tensor decomposion, this intrinsic returns two groups which are the tensor factors of G.

`IsProportional(X, k)`

Given a matrix group G defined over a finite field that admits a tensor decomposition, this intrinsic returns `true` if and only if the matrix X is composed of $k \times k$ blocks which differ only by scalars. If this is indeed the case, the tensor decomposition of X is also returned.

Example H60E4

We define a subgroup of $GL(6, 3)$ which admits a non-trivial tensor decomposition.

```
> P := GL(6, 3);
>
> g := P![ 0, 1, 1, 2, 1, 0, 2, 2, 1, 2, 1, 1, 1, 0, 2, 1, 2, 2, 1, 2, 2,
>          2, 2, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 2, 2 ];
>
> h := P![ 1, 0, 2, 1, 1, 2, 0, 0, 2, 0, 0, 2, 2, 0, 1, 0, 2, 1, 2, 1, 2,
```

```
>          2, 1, 1, 0, 2, 0, 1, 0, 0, 0, 0, 0, 2, 1, 2 ];
>
> G := sub< P | g, h >;
> IsTensor(G);
true
> C := TensorBasis(G);
```

So C is the change-of-basis matrix. If we conjugate $G.1$ by C, we obtain a visible Kronecker product.

```
> G.1^C;
[0 0 2 0 2 0]
[0 0 2 2 2 2]
[2 0 0 0 2 0]
[2 2 0 0 2 2]
[0 0 0 0 1 0]
[0 0 0 0 1 1]
>
```

We use the function `IsProportional` to verify that $G.1^C$ is a Kronecker product.

```
> IsProportional(G.1^C, 2);
true
<
 [2 0]
 [2 2],
 [0 1 1]
 [1 0 1]
 [0 0 2]
>
```

Finally, we display the tensor factors.

```
> A := TensorFactors(G);
> A[1];
MatrixGroup(2, GF(3))
Generators:
    [1 2]
    [2 2]

    [2 0]
    [2 2]
> A[2];
MatrixGroup(3, GF(3))
Generators:
    [0 1 0]
    [1 2 1]
    [1 2 0]

    [0 1 1]
    [1 0 1]
```

```
[0 0 2]
```

60.4.5 Tensor-induced Groups

Let G be a subgroup of $\mathrm{GL}(d, K)$, where $K = \mathbf{F}_q$ and $q = p^e$ for some prime p, and let V be the natural $K[G]$-module. Assume that d has a proper factorisation as u^r. We say that G is tensor-induced if G preserves a decomposition of V as

$$U_1 \otimes U_2 \otimes \cdots \otimes U_r$$

where each U_i has dimension $u > 1$, $r > 1$, and the set of U_i is permuted by G. If G is tensor-induced, then there is a homomorphism of G into the symmetric group S_r.

Theoretical details of the algorithm used may be found in Leedham-Green and O'Brien [LGO02].

`SetVerbose ("TensorInduced", 1)` will provide information on the progress of the algorithm.

`IsTensorInduced(G : parameters)`

`InducedDegree` RngIntElt *Default :* "*All*"

Given a matrix group G defined over a finite field, return `true` if G is tensor-induced, `false` if G is not tensor-induced, and `"unknown"` if no decision can be reached.

If the value of the parameter `InducedDegree` is set to r, then the algorithm will search for homomorphisms into the symmetric group of degree r only. Otherwise is will consider all valid degrees.

`TensorInducedBasis(G)`

Given a matrix group G defined over a finite field that is tensor-induced, this intrinsic returns the change-of-basis matrix which exhibits that G is tensor-induced.

`TensorInducedPermutations(G)`

Given a matrix group G defined over a finite field that is tensor-induced, this intrinsic returns a sequence whose i-th entry is the homomorphic image of $G.i$ in S_r.

`TensorInducedAction(G, g)`

Given a matrix group G defined over a finite field that is tensor-induced, this intrinsic returns the tensor-induced action of the element $g \in G$.

Example H60E5

We illustrate the use of the functions for determining if a matrix group is tensor-induced.

```
> G := GL(2, 3);
> S := Sym(3);
> G := TensorWreathProduct(G, S);
> IsTensorInduced(G);
true
```

We next recover the permutations.

```
> TensorInducedPermutations(G);
[
    Id(S),
    Id(S),
    (1, 2, 3),
    (1, 2)
]
```

Hence $G.1$ and $G.2$ are in the kernel of the homomorphism from G to S. We extract the change-of-basis matrix C and then conjugate $G.1$ by C, thereby obtaining a visible Kronecker product.

```
> C := TensorInducedBasis(G);
> x := G.1^C;
> x;
[2 0 0 0 0 0 0 0]
[0 2 0 0 0 0 0 0]
[0 0 2 0 0 0 0 0]
[0 0 0 2 0 0 0 0]
[1 0 0 0 1 0 0 0]
[0 1 0 0 0 1 0 0]
[0 0 1 0 0 0 1 0]
[0 0 0 1 0 0 0 1]
```

Finally, we verify that $x = G.1^C$ is a Kronecker product for each of 2 and 4.

```
> IsProportional(x, 2);
true
<[2 0]
[0 2], [1 0 0 0]
[0 1 0 0]
[2 0 2 0]
[0 2 0 2]>
> IsProportional(x, 4);
true
<[2 0 0 0]
[0 2 0 0]
[0 0 2 0]
[0 0 0 2], [1 0]
[2 2]>
```

60.4.6 Normalisers of Extraspecial r-groups and Symplectic 2-groups

Let $G \leq GL(d, q)$, where $d = r^m$ for some prime r. If G is contained in the normaliser of an r-group R, of order either r^{2m+1} or 2^{2m+2}, then either R is extraspecial (in the first case), or R is a 2-group of symplectic type (that is, a central product of an extraspecial 2-group with the cyclic group of order 4).

If $d = r$ is an odd prime, we use the Monte Carlo algorithm of Niemeyer [Nie05] to decide whether or not G normalises such a subgroup. Otherwise, the corresponding intrinsic `IsExtraSpecialNormaliser` searches for elements of the normal subgroup, and can only reach a negative conclusion in certain limited cases. If it cannot reach a conclusion it returns `"unknown"`.

`IsExtraSpecialNormaliser(G)`

Given a matrix group G defined over a finite field, the intrinsic returns `true` if G normalises an extraspecial r-group or 2-group of symplectic type, `false` if G is known not to normalise an extraspecial r-group or a 2-group of symplectic type, or `"unknown"` if it cannot reach a conclusion.

`ExtraSpecialParameters(G)`

Given a matrix group G defined over a finite field that is known to normalise an extraspecial r-group or 2-group of symplectic type, this intrinsic returns a sequence of two integers, r and n, where the extraspecial or symplectic subgroup R normalised by G has order r^n.

`ExtraSpecialGroup(G)`

Given a matrix group G defined over a finite field that is known to normalise an extraspecial r-group or 2-group of symplectic type, this intrinsic returns the extraspecial or symplectic subgroup normalised by G.

`ExtraSpecialNormaliser(G)`

Given a matrix group G defined over a finite field that is known to normalise an extraspecial r-group or 2-group of symplectic type, this intrinsic returns the action of the generators of G on its normal extraspecial or symplectic subgroup as a sequence of matrices, each of degree $2r$, one for each generator of G.

`ExtraSpecialAction(G, g)`

Given a matrix group G defined over a finite field that is known to normalise an extraspecial r-group or 2-group of symplectic type, this intrinsic returns a matrix of degree $2r$ describing the action of element g on the extraspecial or symplectic group normalised by G.

`ExtraSpecialBasis(G)`

Given a matrix group G defined over a finite field, which is the odd prime degree case of G normalising an extraspecial r-group or 2-group of symplectic type, this intrinsic returns the change-of-basis matrix which conjugates the normal extraspecial subgroup into a "nice" representation, generated by a diagonal and a permutation matrix.

Example H60E6

For this example we construct a subgroup G of $\mathrm{GL}(7,8)$ that normalises an extraspecial r-group or 2-group of symplectic type.

```
> F:=GF(8);
> P:=GL(7,F);
> w := PrimitiveElement(F);
> g1:=P![
> w,0,w^2,w^5,0,w^3,w,w,1,w^6,w^3,0,w^4,w,w^2,w^6,w^4,1,w^3,w^3,w^5,
> w^6,w,w^3,1,w^5,0,w^4,1,w^6,w^3,w^6,w^3,w^2,w^2,w^3,w^6,w^6,w^4,1,w^2,w^4,
> w^5,w^4,w^2,w^6,1,w^5,w ];
> g2:=P![w^3,w^4,w^2,w^6,w,w,w^3,w^3,w^4,w,w,w^2,w^3,w^3,w,w^3,w^5,w,1,w^3,w,
> 0,w^2,w^6,w,w^5,1,w,w^6,0,w^3,0,w^4,w,w^5,w^3,w^3,1,w^3,w^5,w^5,w^3,
> w^4,w^6,w,w^6,w^4,w^4,0 ];
> g3:=P![w^5,w^6,w^2,w,w,w^4,w^6,w^6,w^6,w,w^6,w,1,w^3,w,w^6,w^2,w,w^6,w^3,w^6,
> w^2,w^6,w^6,w^3,w,w^6,w^5,0,w^4,w^6,w^6,w,w^2,0,w,w^3,w^5,w^2,w^3,w^4,w^6,
> 0,w^3,w,w^3,w^4,w^3,1];
> gens := [g1,g2,g3];
> G := sub< P | gens >;
> IsExtraSpecialNormaliser(G);
true
```

So G has the desired normaliser property.

```
> ExtraSpecialParameters (G);
[ 7, 3 ]
> N:=ExtraSpecialNormaliser(G);
> N;
[
    [3 4]
    [1 4],
    [4 3]
    [0 2],
    [1 0]
    [0 1]
]
```

60.4.7 Writing Representations over Subfields

The algorithm implemented by these functions is due to Glasby, Leedham-Green and O'Brien [GLGO05]. We also provide access to an earlier algorithm for the non-scalar case developed by Glasby and Howlett [GH97].

`IsOverSmallerField(G : parameters)`

`Scalars`	BoolElt	*Default :* `false`
`Algorithm`	MonStgElt	*Default :* "*GLO*"

Given an absolutely irreducible matrix group G defined over a finite field K, this intrinsic decides whether or not G has an equivalent representation over a subfield of K. If so, it returns `true` and the representation over the smallest possible subfield, otherwise it returns `false`. If the optional argument `Scalars` is `true` then decide whether or G modulo scalars has an equivalent representation over a subfield of K. If the optional argument `Algorithm` is set to `"GH"`, then the non-scalar case uses the original Glasby and Howlett algorithm. The default is the Glasby, Leedham-Green and O'Brien algorithm, specified by `"GLO"`.

`IsOverSmallerField(G, k : parameters)`

`Scalars`	BoolElt	*Default :* `false`
`Algorithm`	MonStgElt	*Default :* "*GLO*"

Given an absolutely irreducible matrix group G defined over a finite field K, and a positive integer k which is a proper divisor of the degree of K, this intrinsic decides whether or not G has an equivalent representation over a proper subfield of K having degree k over the prime field. If so, it returns `true` and the representation over this subfield, else it returns `false`. If the optional argument `Scalars` is `true` then it decides whether or not G modulo scalars has an equivalent representation over a degree k subfield of K. If the optional argument `Algorithm` is set to `"GH"`, then the non-scalar case uses the original Glasby and Howlett algorithm. The default is the Glasby, Leedham-Green and O'Brien algorithm, specified by `"GLO"`.

`SmallerField(G)`

Given an absolutely irreducible matrix group G defined over a finite field K, which can be written over a proper subfield of K (possibly modulo scalars), return the subfield.

`SmallerFieldBasis(G)`

Given an absolutely irreducible matrix group G defined over a finite field K, which can be written over a proper subfield of K (possibly modulo scalars), return the change of basis matrix for G which rewrites G over the smaller field.

`SmallerFieldImage(G, g)`

Given an absolutely irreducible matrix group G defined over a finite field K, which can be written over a proper subfield of K (possibly modulo scalars), return the image of $g \in G$ in the group defined over the subfield.

Example H60E7

We define a subgroup of $\mathrm{GL}(3,8)$ which can be written over $\mathbf{F}_2$.

```
> G := GL (2, GF (3, 2));
> H := GL (2, GF (3, 8));
> K := sub < H | G.1, G.2 >;
> K;
MatrixGroup(2, GF(3^8))
Generators:
    [ $.1^820        0]
    [       0        1]

    [       2        1]
    [       2        0]
> flag, M := IsOverSmallerField (K);
> flag;
true
> M;
MatrixGroup(2, GF(3^2))
Generators:
    [$.1^7 $.1^2]
    [    1     2]

    [$.1^7 $.1^6]
    [$.1^2 $.1^5]
> F := GF(3, 4);
> G := MatrixGroup<2, F | [ F.1^52, F.1^72, F.1^32, 0 ],
>                                 [ 1, 0, F.1^20, 2 ] >;
> flag, X := IsOverSmallerField (G);
> flag;
false
```

We now see if G has an equivalent representation modulo scalars.

```
> flag, X := IsOverSmallerField (G: Scalars := true);
> flag;
true
> X;
MatrixGroup(2, GF(3))
Generators:
    [2 1]
    [1 0]
    [2 1]
    [1 1]
> SmallerField (G);
Finite field of size 3
> SmallerFieldBasis (G);
[F.1^33 F.1^23]
[F.1^43 F.1^63]
```

```
> g := G.1 * G.2^2; g;
[F.1^52 F.1^72]
[F.1^32      0]
> SmallerFieldImage (G, g);
[1 2]
[2 0]
```

`WriteOverSmallerField(G, F)`

Given a group G of $d \times d$ matrices over a finite field E having degree e and a subfield F of E having degree f, write the matrices of G as de/f by de/f matrices over F and return the group and the isomorphism.

Example H60E8

We define the group $\mathrm{GL}(2, 4)$ and then rewrite in over $\mathbf{F}_2$ as a degree 4 matrix group.

```
> G := GL(2, 4);
> H := WriteOverSmallerField(G, GF(2));
> H;
MatrixGroup(4, GF(2))
Generators:
    [0 1 0 0]
    [1 1 0 0]
    [0 0 1 0]
    [0 0 0 1]

    [1 0 1 0]
    [0 1 0 1]
    [1 0 0 0]
    [0 1 0 0]
```

60.4.8 Decompositions with Respect to a Normal Subgroup

`SearchForDecomposition(G, S)`

Given a matrix group G defined over a finite field and a sequence S of elements of G, this intrinsic first constructs the normal closure N of S in G. It then seeks to decide whether or not G, with respect to N, has a decomposition corresponding to one of the categories (ii)–(vi) in the theorem of Aschbacher stated at the beginning of this section. Theoretical details of the algorithms used may be found in Holt, Leedham-Green, O'Brien, and Rees [HLGOR96a].

In summary, it tests for one of the following possibilities:

(ii) G acts semilinearly over an extension field L of K, and N acts linearly over L;

(iii) G acts imprimitively on V and N fixes each block of imprimitivity;

(iv) G preserves a tensor product decomposition $U \otimes W$ of V, where N acts as scalar matrices on U;

(v) N acts absolutely irreducibly on V and is an extraspecial p-group for some prime p, or a 2-group of symplectic type;

(vi) G preserves a tensor-induced decomposition $V = \otimes^m U$ of V for some $m > 1$, where N acts absolutely irreducibly on V and fixes each of the m factors.

If one of the listed decompositions is found, then the function *reports* the type found and returns `true`; if no decomposition is found with respect to N, then the function returns `false`. The answer provided by the function is conclusive for decompositions of types (ii)–(v), but a negative answer for (vi) is not necessarily conclusive.

Each test involves a decomposition of G with respect to the normal subgroup N (which may sometimes be trivial or scalar). In (ii), N is the subgroup of G acting linearly over the extension field irreducibly on V. In (iii), N is the subgroup which fixes each of the subspaces in the imprimitive decomposition of V. In (iv), it is the subgroup acting as scalar matrices on one of the factors in the tensor-product decomposition. In (v), N is already described, and in (vi), it is the subgroup fixing each of the factors in the tensor-induced decomposition (so N itself falls in Category (iv)).

If any one of these decompositions can be found, then it may be possible to obtain an explicit representation of G/N and hence reduce the study of G to a smaller problem. For example, in Category (iii), G/N acts as a permutation group on the subspaces in the imprimitive decomposition of V. Currently only limited facilities are provided to construct G/N.

Information about the progress of the algorithm can be output by setting the verbose flag `SetVerbose ("Smash", 1)`.

60.4.8.1 Accessing the Decomposition Information

The access functions described in the sections on Primitivity Testing, Semilinearity, Tensor Products, Tensor Induction, and Normalisers of extraspecial groups may be used to extract information about decompositions of type (ii), (iii), (iv), (v) and (vi). We illustrate such decompositions below.

Example H60E9

We begin with an example where no decomposition exists.

```
> G := GL(4, 5);
> SearchForDecomposition (G, [G.1]);
Smash: No decomposition found
false
```

The second example is of an imprimitive decomposition.

```
> M := GL (4, 7);
> P := Sym (3);
> G := WreathProduct (M, P);
> SearchForDecomposition (G, [G.1, G.2]);
Smash: G is imprimitive
true
> IsPrimitive (G);
false
> BlocksImage (G);
Permutation group acting on a set of cardinality 3
    Id($)
    Id($)
    (1, 2, 3)
    (1, 2)
```

The third example admits a semilinear decomposition.

```
> P := GL(6,3);
> g1 := P![0,1,0,0,0,0,-1,0,0,0,0,0,
>          0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1];
> g2 := P![-1,0,0,0,1,0,0,-1,0,0,0,1,
>          -1,0,0,0,0,0,0,-1,0,0,0,0,0,0,-1,0,0,0,0,0,0,-1,0,0];
> g3 := P![1,0,0,0,0,0,0,-1,0,0,0,0,
>          0,0,1,0,0,0,0,0,0,-1,0,0,0,0,0,0,1,0,0,0,0,0,0,-1];
> G := sub <P | g1, g2, g3 >;
>
> SearchForDecomposition (G, [g1]);
Smash: G is semilinear
true
> IsSemiLinear (G);
true
> DegreeOfFieldExtension (G);
2
```

```
> CentralisingMatrix (G);
[2 2 0 0 0 0]
[1 2 0 0 0 0]
[0 0 2 2 0 0]
[0 0 1 2 0 0]
[0 0 0 0 2 2]
[0 0 0 0 1 2]
> FrobeniusAutomorphisms (G);
[ 1, 1, 3 ]
```

The fourth example admits a tensor product decomposition.

```
> F := GF(5);
> G := GL(5, F);
> H := GL(3, F);
> P := GL(15, F);
> A := MatrixAlgebra (F, 5);
> B := MatrixAlgebra (F, 3);
> g1 := A!G.1; g2 := A!G.2;  g3 := A!Identity(G);
> h1 := B!H.1; h2 := B!H.2; h3 := B!Identity(H);
> w := TensorProduct (g1, h3);
> x := TensorProduct (g2, h3);
> y := TensorProduct (g3, h1);
> z := TensorProduct (g3, h2);
> G := sub < P | w, x, y, z>;
> SearchForDecomposition (G, [G.1, G.2]);
Smash: G is a tensor product
true
> IsTensor (G);
true
> TensorBasis (G);
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[4 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 4 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 4 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 4 0 1]
[0 0 0 0 0 0 0 0 0 4 0 1 0 0 0]
[1 4 4 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 4 4 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 4 4 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 4 4]
[0 0 0 0 0 0 0 0 0 1 4 4 0 0 0]
```

Our fifth example is of a tensor-induced decomposition.

```
> M := GL (3, GF(2));
```

```
> P := Sym (3);
> G := TensorWreathProduct (M, P);
> SearchForDecomposition (G, [G.1]);
Smash: G is tensor induced
true
>
> IsTensorInduced (G);
true
> TensorInducedPermutations (G);
[ Id(P), Id(P), (1, 3, 2), (1, 3) ]
```

Our final example is of a normaliser of a symplectic group.

```
> F := GF(5);
> P := GL(4,F);
> g1 := P![ 1,0,0,0,0,4,0,0,2,0,2,3,3,0,4,3];
> g2 := P![ 4,0,0,1,2,4,4,0,1,0,1,2,0,0,0,1];
> g3 := P![ 4,0,1,1,0,1,0,0,0,1,3,4,0,4,3,2];
> g4 := P![ 2,0,4,3,4,4,2,4,0,1,3,4,4,2,0,1];
> g5 := P![ 1,1,3,4,0,0,3,4,2,0,0,4,3,1,3,4];
> g6 := P![ 2,0,0,0,0,2,0,0,0,0,2,0,0,0,0,2];
> G := sub < P | g1, g2, g3, g4, g5, g6 >;
> SearchForDecomposition (G, [G.4]);
Smash: G is normaliser of symplectic 2-group
true
> IsExtraSpecialNormaliser (G);
true
> ExtraSpecialParameters (G);
[2, 6]
> g := G.1 * G.2;
> ExtraSpecialAction(G, g);
[0 1 0 0]
[1 1 0 0]
[0 1 1 1]
[1 1 1 0]
```

60.5 Constructive Recognition for Simple Groups

For each finite non-abelian simple group S, we designate a specific *standard copy* of S. The standard copy has a designated set of *standard generators*. For example, the standard copy of $\mathrm{Alt}(n)$ is on n points; its standard generators are $(1,2,3)$ and either of $(3,\ldots,n)$ or $(1,2)(3,\ldots,n)$ according to the parity of n. For a projective representation, the standard copy is the quotient of a matrix group by its scalar subgroup. For example, the standard copy of $\mathrm{PSL}(n,q)$ is the quotient of $\mathrm{SL}(n,q)$ by its scalar subgroup.

To compute in a copy G of S, we first construct effective isomorphisms between G and its standard copy. We do this by finding generators in G that correspond to the standard generators of S under an isomorphism. More formally, a *constructive recognition algorithm* for a non-abelian simple group G (possibly with decorations) solves the following problem: construct an isomorphism ϕ from G to a standard copy S of G, such that $\phi(g)$ can be computed efficiently for every $g \in G$. This is done by constructing standard generators in both G and its standard copy S.

A *rewriting algorithm* for G solves the *constructive membership problem*: given $g \in U \geq G = \langle X \rangle$, decide whether or not $g \in G$, and if so express g as an SLP in X. (Here U is the generic overgroup of G, such as $\mathrm{GL}(d,q)$ or $\mathrm{Sym}(n)$.) The rewriting algorithm is used to make the isomorphism between S and G effective. To compute the image of an arbitrary element s of S in G, we first write s as an SLP in the standard generators of S and then evaluate the SLP in the copy of the standard generators in G.

To verify that the homomorphism from S to G is an isomorphism, we can evaluate in G a *standard presentation* for S on its standard generators. If the copy of the standard generators in G satisfy the presentation, then we have proved that we have an isomorphism.

For a detailed discussion of these topics, see [O'B11, LGO09].

`ClassicalStandardGenerators(type, d, q)`

This intrinsic produces the standard generators of Leedham-Green and O'Brien for the quasisimple classical group of specified type in dimension d over field of size q. The type is designated by the argument "type" which must be one of the strings "SL", "Sp", "SU", "Omega", "Omega-", or "Omega+". The standard generators defined a specific copy of a classical group and are defined in [LGO09] and [DLLGO13].

`ClassicalConstructiveRecognition(G : parameters)`

`Case`	MonStgElt	*Default :* "*unknown*"
`Randomiser`	GrpRandProc	*Default :*

The argument G must be a matrix group defined over a finite field such that $G = \langle X \rangle$ is conjugate to a quasisimple classical group in its natural representation in dimension at least 2. The intrinsic constructs a copy $\mathcal{S}$ in G of the generators defined by `StandardGenerators`. If G is quasisimple and classical, then the function returns

`true`, the standard generators $\mathcal{S}$, and SLPs for these in X; otherwise it returns `false`.

The result returned by the intrinsic `ClassicalType (G)` can be supplied using the optional parameter `Case`.

The parameter `Randomiser` allows the user to specify the random process that is be used to construct random elements and the SLPs returned for the standard generators are in the word group of this process. The default value of `Randomiser` is `RandomProcessWithWords(G)`.

The implementations for even and odd characteristic were developed by Heiko Dietrich and Eamonn O'Brien respectively.

`ClassicalChangeOfBasis(G)`

G is a classical group in its natural representation; return a change-of-basis matrix to conjugate the generators returned by `ClassicalStandardGenerators` to those returned by `ClassicalConstructiveRecognition` (G). The latter intrinsic must have been applied to G.

`ClassicalRewrite(G, gens, type, dim, q, g` : *parameters*`)`

`Method` MonStgElt *Default : "choose"*

Let G be a classical group of type `type`, which is one of the strings "SL", "Sp", "SU", "Omega", "Omega-", or "Omega+", with dimension `dim` over the field $\mathbf{F}_q$ generated by `gens` which satisfy `ClassicalStandardPresentation (type, dim, q)`. Further, let g be an element of `Generic`(G).

If $g \in G$, then the function returns `true` and an SLP for g in `gens`; if $g \notin G$ then the function searches for an SLP w such that g·`Evaluate`$(w, gens)^{-1}$ centralizes G; if it is successful, it returns `false` and w. Otherwise the function returns `false, false`.

The function chooses one of the following methods:

(i) If G is in its natural representation and `gens` is `ClassicalStandardGenerators (type, dim, q)` then Elliot Costi's implementation [Cos09] is used.

(ii) If (i) is not valid, but G is an absolutely irreducible representation in the defining characteristic, then another implementation developed by Csaba Schneider is used. This implementation is based on the description given by Costi [Cos09].

(iii) If neither of (i) and (ii) is valid, then a "black-box" method, independent of the representation of G, developed by Csaba Schneider is used. This case is not yet implemented for orthogonal groups.

The optional parameter `Method` can be used to override the default choice of method. The possible values of `Method` are `CharP` and `BB`.

A description of the algorithm used in the defining characteristic case appears in [Cos09]; a short description of the black-box algorithm appears in [AMPS10]. The code was prepared for distribution by Csaba Schneider.

`ClassicalRewriteNatural(type, CB, g)`

This is a faster specialized verson of the intrinsic `ClassicalRewrite` discussed above; it is designed for classical groups in their **natural representation**.

The argument `type` must be one of the strings "SL", "Sp", "SU", "Omega", "Omega-", or "Omega+". Both `CB` and g are elements of some $\mathrm{GL}(d, q)$.

If g is a member of the group generated by `ClassicalStandardGenerators (type, d, q)`$^{\mathbf{CB}}$ then the function returns `true` and an SLP w such that `Evaluate (w, ClassicalStandardGenerators (type, d, q)`$^{\mathbf{CB}}$`)` $= g$. Otherwise the function returns `false, false`.

This algorithm was developed and implemented by Elliot Costi; the code was prepared for distribution by Csaba Schneider.

`ClassicalStandardPresentation(type, d, q : parameters)`

`Projective` BoolElt *Default :* `false`

Given the specification `type, d, q` of a quasisimple group G, this intrinsic constructs a presentation on the standard generators for G. The string `type` must be one of "SL", "Sp", "SU", "Omega", "Omega-", or "Omega+", while d is the dimension and q is the cardinality of the finite field. The presentations are described in [LGO]. The relations are returned as SLPs together with the parent SLPGroup.

If the parameter `Projective` is set to true, the intrinsic constructs a presentation for the corresponding projective group.

Example H60E10

As our first illustration, we produce standard generators for $\mathrm{SL}(6, 5^3)$:

```
> E := ClassicalStandardGenerators ("SL", 6, 5^3);
> E;
[
    [      0       1       0       0       0       0]
    [      4       0       0       0       0       0]
    [      0       0       1       0       0       0]
    [      0       0       0       1       0       0]
    [      0       0       0       0       1       0]
    [      0       0       0       0       0       1],

    [      0       0       0       0       0       1]
    [      4       0       0       0       0       0]
    [      0       4       0       0       0       0]
    [      0       0       4       0       0       0]
    [      0       0       0       4       0       0]
    [      0       0       0       0       4       0],

    [      1       1       0       0       0       0]
    [      0       1       0       0       0       0]
    [      0       0       1       0       0       0]
```

```
    [      0       0       0       1       0       0]
    [      0       0       0       0       1       0]
    [      0       0       0       0       0       1],

    [    $.1       0       0       0       0       0]
    [      0 $.1^123       0       0       0       0]
    [      0       0       1       0       0       0]
    [      0       0       0       1       0       0]
    [      0       0       0       0       1       0]
    [      0       0       0       0       0       1]
]
```

We now perform constructive recognition on $\mathrm{SL}(6, 5^3)$, and so obtain S, conjugate to E in $\mathrm{GL}(6, 5^3)$. Observe that the change-of-basis matrix returned by `ClassicalChangeOfBasis` performs this conjugation.

```
> G := SL (6, 5^3);
> f, S, W := ClassicalConstructiveRecognition (G);
> f;
true
> CB := ClassicalChangeOfBasis (G);
> E^CB eq S;
true
```

Note that W is list of SLPs expressing S in terms of defining generators of G.

```
> S eq Evaluate (W, [G.i: i in [1..Ngens (G)]]);
true
```

We next express a random element of G as a SLP in S and then check that the standard generators satisfy the standard presentation.

```
> g := Random (G);
> f, w := ClassicalRewriteNatural ("SL", CB, g);
> Evaluate (w, S) eq g;
true
>
> P, R := ClassicalStandardPresentation ("SL", 6, 5^3);
> Set (Evaluate (R, S));
{
    [      1       0       0       0       0       0]
    [      0       1       0       0       0       0]
    [      0       0       1       0       0       0]
    [      0       0       0       1       0       0]
    [      0       0       0       0       1       0]
    [      0       0       0       0       0       1]
}
```

We perform constructive recognition on a random conjugate of $\mathrm{Sp}(10, 3^6)$ and again check that the standard generators satisfy the standard presentation.

```
> G := RandomConjugate (Sp(10, 3^6));
```

```
> f, S, W := ClassicalConstructiveRecognition (G);
> f;
true
```

The return variable W is list of SLPs expressing S in terms of defining generators of G.

```
> S eq Evaluate (W, [G.i: i in [1..Ngens (G)]]);
true
>
> g := Random (G);
> CB := ClassicalChangeOfBasis (G);
> f, w := ClassicalRewriteNatural ("Sp", CB, g);
> Evaluate (w, S) eq g;
true
>
> P, R := ClassicalStandardPresentation ("Sp", 10, 3^6);
> Set (Evaluate (R, S));
{
    [1 0 0 0 0 0 0 0 0 0]
    [0 1 0 0 0 0 0 0 0 0]
    [0 0 1 0 0 0 0 0 0 0]
    [0 0 0 1 0 0 0 0 0 0]
    [0 0 0 0 1 0 0 0 0 0]
    [0 0 0 0 0 1 0 0 0 0]
    [0 0 0 0 0 0 1 0 0 0]
    [0 0 0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 0 0 1 0]
    [0 0 0 0 0 0 0 0 0 1]
}
```

As another demonstration, we constructively recognise $\Omega^-(16, 2^6)$.

```
> G := RandomConjugate (OmegaMinus (16, 2^6));
> f, S, W := ClassicalConstructiveRecognition (G);
> f;
true
```

A random element of G is expressed as an SLP in S:

```
> g := Random (G);
> CB := ClassicalChangeOfBasis (G);
> f, w := ClassicalRewriteNatural ("Omega-", CB, g);
> Evaluate (w, S) eq g;
true
```

Finally, we illustrate using `ClassicalRewrite` to write an element of a classical group in a non-natural representation as an SLP in its standard generators.

```
> gens := [ExteriorSquare (x) : x in ClassicalStandardGenerators ("Sp", 6, 25)];
> G := sub<Universe (gens) | gens>;
> x := Random (G);
```

```
> f, w := ClassicalRewrite (G, gens, "Sp", 6, 25, x);
> f;
true
> Evaluate (w, gens) eq x;
true
> f, w := ClassicalRewrite (G, gens, "Sp", 6, 25, x : Method := "BB");
> f;
true
> Evaluate (w, gens) eq x;
true
```

60.6 Composition Trees for Matrix Groups

A *composition tree* for a group G can be viewed as a data structure that present a group in terms of its composition factors. The tree is constructed recursively and the data structure facilitates the rewriting of elements of G in terms of different generating sets.

The basic strategy for computing a *composition tree* of a matrix group employs a combination of a constructive version of Aschbacher's theorem [Asc84] and constructive recognition algorithms for finite simple groups. The basic algorithms are described in [LG01, O'B06, O'B11] while some new ideas introduced in [NS06] are incorporated. A detailed account of the entire *CompositionTree* procedure appears in [BHLGO11].

The algorithm to construct a composition tree proceeds as follows. Given a group G, either:

(i) Construct an effective homomorphism $\phi : G \to G_1$, for some group G_1. We call ϕ a *reduction* since G_1 is "smaller" than G in some respect – for example, its degree or field of definition.

(ii) Or deduce that G is cyclic, elementary abelian, or "close" to being non-abelian simple. Now G becomes a *leaf* in the tree.

Assume that Case (i) applies.

1. Now construct a composition tree for G_1.
2. Construct generators for $G_0 := \mathrm{Ker}(\phi)$. This requires a rewriting algorithm for G_1.
3. Construct a composition tree for G_0.
4. Combine the composition trees for G_1 and G_0 into a tree for G.

If $G \leq \mathrm{GL}(d, q)$, then we exploit Aschbacher's theorem [Asc84] in Step (1). This requires algorithms to decide if G lies in a certain Aschbacher class, and to construct the corresponding ϕ. Other homomorphisms, such as the determinant map, may also be used.

The group associated with a leaf need not be simple. It may be cyclic or elementary abelian, a soluble or non-abelian simple primitive permutation group, or an absolutely irreducible matrix group that is simple modulo its centre. The decisions on just which

groups are treated as a leaves are partly dictated by complexity considerations, and partly based on the quality of available algorithms to process a leaf. For example, we observe no practical advantage flowing from refining a cyclic group to its composition factors.

Once a composition tree for $G = \langle X \rangle$ has been constructed, then a second list Y of *nice generators* are stored with G. We call $\langle Y \rangle$ the *nice group*. The intrinsic `CompositionTree` constructs the nice generators Y as SLPs in X. The rewriting algorithm solves rewriting problems on Y and the resulting SLPs can then be rewritten to provide SLPs on X.

The verbose flag `SetVerbose("CompositionTree", n)` with $n = 1, \ldots, 10$ may be used to print increasing levels of information on the progress of the functions.

`CompositionTree(G : parameters)`

`Verify`	BoolElt	*Default :* `false`
`Scalar`	FldFinElt	*Default :* 1
`KernelBatchSize`	RngIntElt	*Default :* 5
`MandarinBatchSize`	RngIntElt	*Default :* 100
`MaxHomFinderFails`	RngIntElt	*Default :* 1
`MaxQuotientOrder`	RngIntElt	*Default :* 10^6
`FastTensorTest`	BoolElt	*Default :* `true`
`MaxBSGSVerifyOrder`	RngIntElt	*Default :* 2000
`AnalysePermGroups`	BoolElt	*Default :* `false`
`KnownLeaf`	BoolElt	*Default :* `false`
`NamingElements`	RngIntElt	*Default :* 200
`UnipotentBatchSize`	RngIntElt	*Default :* 100
`PresentationKernel`	BoolElt	*Default :* `true`

Given a matrix group G defined over a finite field, this intrinsic constructs a composition tree for G and returns the tree.

`Verify`: If `true`, then verify correctness of the tree during construction.

`KernelBatchSize`: The number of normal generators used to construct the kernel of homomorphism.

`MandarinBatchSize`: The number of random elements used to check correctness of the outcome of Monte-Carlo algorithms.

`MaxHomFinderFails`: Assume a negative answer after this many failures of certain Monte Carlo algorithms.

`AnalysePermGroups`: If `false`, then always treat the permutation group as a leaf, and do not analyse its structure.

`NamingElements`: The number of random elements used in calls to `LieType` and `RecogniseClassical`.

`MaxQuotientOrder`: A leaf with larger order will not be fully refined to its composition factors.

`FastTensorTest`: Use only the fast tensor product test.

`MaxBSGSVerifyOrder`: If `RandomSchreier` is used on a leaf and it has order less than `MaxBSGSVerifyOrder`, then `Verify` the calculation.

`PresentationKernel`: Use presentations to obtain kernels, where possible.

`UnipotentBatchSize`: Batch size for the unipotent kernels.

`CompositionTreeFastVerification(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. The intrinsic determines if correctness of the composition tree can be verified at modest cost using presentations. In effect, the intrinsic determines whether presentations on nice generators are known for all the leaves.

`CompositionTreeVerify(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. The intrinsic verifies the correctness of the composition tree by using it to construct a presentation for G. If G satisfies the presentation, then return `true`, and the relators of the presentation as SLPs; otherwise return `false`. The presentation is on the group returned by `CompositionTreeNiceGroup`(G).

`CompositionTreeNiceGroup(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. The intrinsic returns the nice group for G.

`CompositionTreeSLPGroup(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure and the associated nice group H has previously been constructed. The intrinsic returns the word group W for H, and the map from W to H.

`DisplayCompTreeNodes(G :` *parameters*`)`

`NonTrivial`	BoolElt	*Default :* `true`
`Leaves`	BoolElt	*Default :* `false`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic displays information about the nodes in the composition tree for G. The tree is traversed in-order. If parameter `NonTrivial` is `true`, then only non-trivial nodes will be displayed. If parameter `Leaves` is `true` then only leaves will be displayed.

`CompositionTreeNiceToUser(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns the coercion map from SLPs in nice generators of G to SLPs in input user generators of G, as well as the SLPs of the nice generators given in terms the user generators.

`CompositionTreeOrder(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns the order of G.

`CompositionTreeElementToWord(G, g)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. Given an element $g \in G$, return `true` and an SLP for g in the nice generators of G, otherwise return `false`.

`CompositionTreeCBM(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns a change-of-basis matrix that exhibits the Aschbacher reductions of G given by the composition tree.

`CompositionTreeReductionInfo(G, t)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns a string description of the reduction at the internal node t in the composition tree for G, as well as the image and kernel of this reduction.

`CompositionTreeSeries(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns:

1. A normal series of subgroups $1 = G_0 < G_1 < ... < G_k = G$.
2. Maps $G_i \mapsto S_i$, where S_i is the standard copy of G_i/G_{i-1}, where $i \geq 1$. The kernel of this map is G_{i-1}. Observe that S_i may be the standard copy plus scalars Z, and the map is then a homomorphism modulo scalars, so that the kernel is $(G_{i-1}.Z)/Z$.
3. Maps $S_i \mapsto G_i$.
4. Maps $S_i \mapsto$ `WordGroup`(S_i).
5. Boolean flag `true` or `false` to indicate if the series is a true composition series.
6. A sequence of the leaf nodes in the composition tree corresponding to each composition factor. All maps are defined by rules, so `Function` can be applied on them to avoid built-in membership testing.

`CompositionTreeFactorNumber(G, g)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns the minimal integer i such that g lies in the ith-term of the normal series returned by `CompositionTreeSeries` for G.

`HasCompositionTree(G)`

Given a matrix group G defined over a finite field, this intrinsic returns `true` if G has a composition tree and `false` otherwise.

`CleanCompositionTree(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic removes all data related to the composition tree datastructure for G.

Example H60E11

We construct a composition tree for the conformal orthogonal group G of plus type of degree 4 over $\mathbf{F}_{5^2}$.

```
> G := CGOPlus(4, 5^2);
>
> T := CompositionTree(G);
>
> DisplayCompTreeNodes (G: Leaves:=true);
node = 2
parent = 1
depth = 1
scalar = 1
info = leaf, GrpAb, cyclic group, order 12
fast verify = true
----------
node = 4
parent = 3
depth = 2
scalar = 1
info = leaf, GrpAb, cyclic group, order 4
fast verify = true
----------
node = 7
parent = 6
depth = 4
scalar = 12
info = leaf, GrpAb, cyclic group, order 24
fast verify = true
----------
node = 8
parent = 6
depth = 4
scalar = 2
info = leaf, GrpMat, almost simple, <"A", 1, 25>
fast verify = true
----------
node = 9
```

```
parent = 5
depth = 3
scalar = 1
info = leaf, GrpMat, almost simple, <"A", 1, 25>
fast verify = true
----------
```

We now verify correctness of the composition tree. In order to show what is going on we illustrate various pieces of the verification process. We begin by setting up the nice group H for G and its associated SLP group; observe that $H = G$. After checking that verification can be done quickly, we perform the verification.

```
> H := CompositionTreeNiceGroup(G);
> W := CompositionTreeSLPGroup(G);
>
> CompositionTreeFastVerification(G);
true
>
> f, R := CompositionTreeVerify(G);
> #R;
73
```

At this point we have verified correctness. However, we now explicitly evaluate the relations R on the generators of H. This step has already been performed by `CompositionTreeVerify` so it is shown here just for demonstration purposes.

```
> Set(Evaluate(R, [H.i:i in [1..Ngens(H)]]));
{
    [     1      0      0      0]
    [     0      1      0      0]
    [     0      0      1      0]
    [     0      0      0      1]
}
```

Now that we know that the composition tree is correct, we ask for the order of G.

```
> CompositionTreeOrder(G);
11681280000
```

Express the element g of G as a SLP on the generators of the nice group H.

```
> g := Random(G);
> f, w := CompositionTreeElementToWord(G, g);
> Evaluate(w, [H.i:i in [1..Ngens(H)]]) eq g;
true
```

Rewrite the SLP in terms of the user-supplied generators for G.

```
> tau := CompositionTreeNiceToUser(G);
> tau;
```

```
Mapping from: GrpSLP: W to SLPGroup(5)
```

Images of elements of W under *tau* lie in `WordGroup`(G).

```
> v := tau(w);
> Evaluate (v, [G.i : i in [1..Ngens(G)]]) eq g;
true
```

Test a random element of the generic group for G for membership. (The generic group will be the general linear group $\mathrm{GL}(4, 5^2)$.)

```
> x := Random(Generic(G));
> f, w := CompositionTreeElementToWord(G, x);
> f;
false
```

Finally, we construct a normal series for G and locate a random element within this series.

```
> CS, _, _, _, flag := CompositionTreeSeries(G);
> "Series is composition series? ", flag;
Series is composition series?  true
> "Length is ", #CS;
Length is  10
>
> g := Random(G);
> CompositionTreeFactorNumber(G, g);
10
```

Example H60E12

In this example, we choose a maximal subgroup of the linear group $SL(10, 2^8)$ and compute its composition tree.

```
> X := ClassicalMaximals ("L", 10, 2^8);
> G := X[1];
>
> T := CompositionTree (G);
>
> DisplayCompTreeNodes (G: Leaves:=true, NonTrivial:=true);
node = 6
parent = 5
depth = 5
scalar = 1
info = leaf, GrpAb, cyclic group, order 255
fast verify = true
----------
node = 9
parent = 8
depth = 5
scalar = 1
info = leaf, GrpMat, almost simple, <"A", 8, 256>
```

```
fast verify = true
----------
node = 13
parent = 12
depth = 3
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 15
parent = 14
depth = 4
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 17
parent = 16
depth = 5
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 19
parent = 18
depth = 6
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 21
parent = 20
depth = 7
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 23
parent = 22
depth = 8
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 25
parent = 24
depth = 9
scalar = 1
```

```
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 27
parent = 26
depth = 10
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 29
parent = 28
depth = 11
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
```

We now set up the nice group H for G and its associated SLP group; observe that $H = G$.

```
> H := CompositionTreeNiceGroup(G);
> "# of generators of H is ", Ngens(H);
# of generators of H is  77
> W := CompositionTreeSLPGroup(G);
```

After checking that correctness of the composition tree can be verified quickly, we perform verification.

```
> CompositionTreeFastVerification(G);
true
> f, R := CompositionTreeVerify(G);
> #R;
3028
```

Evaluate the relations on the generators of H.

```
> Set (Evaluate (R, [H.i:i in [1..Ngens (H)]]));
{
    [1 0 0 0 0 0 0 0 0 0]
    [0 1 0 0 0 0 0 0 0 0]
    [0 0 1 0 0 0 0 0 0 0]
    [0 0 0 1 0 0 0 0 0 0]
    [0 0 0 0 1 0 0 0 0 0]
    [0 0 0 0 0 1 0 0 0 0]
    [0 0 0 0 0 0 1 0 0 0]
    [0 0 0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 0 0 1 0]
    [0 0 0 0 0 0 0 0 0 1]
```

```
}
```

Express the element g of G as a SLP on the generators of the nice group H. Then rewrite the SLP in terms of user generators for G.

```
> g := Random (G);
> f, w := CompositionTreeElementToWord (G, g);
> Evaluate (w, [H.i:i in [1..Ngens (H)]]) eq g;
true
>
> tau := CompositionTreeNiceToUser (G);
> tau;
Mapping from: GrpSLP: W to SLPGroup(4)
>
> v := tau (w);
> Evaluate (v, [G.i : i in [1..Ngens (G)]]) eq g;
true
```

Next test a random element of the generic group of G for membership of G.

```
> x := Random (Generic (G));
> f, w := CompositionTreeElementToWord (G, x);
> f;
false
```

Finally, we construct a normal series for G.

```
> CS, _, _, _, flag := CompositionTreeSeries (G);
> "Series is composition series? ", flag;
Series is composition series?  true
> "Length is ", #CS;
Length is  78
```

60.7 The LMG functions

The LMG (large matrix group) functions are designed to provide a user-friendly interface to the `CompositionTree` package, and thereby enable the user to carry out a limited range of structural calculations in a matrix group that is too large for the use of BSGS methods.

By default, these methods have a small probability of failing or even of returning incorrect results. For most examples, at the cost of some extra time, the user can ensure that the computed results are verified as correct by calling `LMGInitialise` with the `Verify` flag on the group G before calling any of the other functions. (Once `CompositionTree` or any of the LMG functions has been called on a group, further calls of `LMGInitialise` will have no effect.)

Let G be a matrix group over a finite field. On the first call of any of the LMG functions on G, MAGMA decides whether it will use BSGS or Composition Tree based methods on G. It does this by carrying out a quick calculation to decide whether any of the basic orbit lengths would be larger than a constant `LMGSchreierBound`, which is

set to 40000 by default, but can be changed by the user. If all basic orbit lengths are at most `LMGSchreierBound`, then BSGS methods are used on G, and the LMG functions are executed using the corresponding standard MAGMA functions. Otherwise, Composition Tree methods are used, starting with a call of `CompositionTree`(G).

Composition Tree methods are also used if the user calls `CompositionTree` on the group before using th LMG functions, or if the user calls `LMGInitialize` with the `Al` option set to `"CompositionTree"`.

`SetVerbose("LMG", n)` with $n = 1, 2$ or 3 will provide increasing levels of information on the progress of the functions.

`SetLMGSchreierBound(n)`

Set the constant `LMGSchreierBound` to n.

`LMGInitialize(G : parameters)`

`LMGInitialise(G : parameters)`

`Al`	MONSTGELT	*Default :* ""
`Verify`	BOOLELT	*Default :* `false`
`RandomSchreierBound`	RNGINTELT	*Default :* `LMGSchreierBound`

It is not normally necessary to call this function but, by setting the optional parameters, it can be used to initialise G for LMG computations with a different value of `LMGSchreierBound` or, by setting `Al` to be `"CompositionTree"` (or `"CT"` or `"RandomSchreier"` (or `"RS"`), to force the use of either Composition Tree or BSGS methods on G.

If the `Verify` flag is set, then an attempt will be made to verify the correctness of the computed BSGS or Composition Tree. This will make the initialisation process slower, and for some groups the increased memory requirements will make verification impractical. In that case, a warning message is displayed.

`LMGOrder(G)`

Given a matrix group G defined over a finite field, this intrinsic returns the order of G.

`LMGFactoredOrder(G)`

Given a matrix group G defined over a finite field, this intrinsic returns the factored order of G.

`LMGIsIn(G, x)`

Given a matrix group G defined over a finite field $\mathbf{F}_{n,q}$ of G, the intrinsic returns `true` if x is in G and `false` otherwise.

`LMGIsSubgroup(G, H)`

Given a matrix group G defined over a finite field $\mathbf{F}_{n,q}$ of G, the intrinsic returns `true` if $H \leq G$ and `false` otherwise.

`LMGEqual(G, H)`

Given a matrix groups G and H belonging to a common overgroup $\mathrm{GL}(n, q)$, the intrinsic returns `true` if G and H are equal and `false` otherwise.

`LMGIndex(G, H)`

Given a matrix group G defined over a finite field, and a subgroup H of G, the intrinsic returns the index of H in G.

`LMGIsNormal(G, H)`

Given a matrix group G defined over a finite field, and a subgroup H of G, the intrinsic returns `true` if H is normal in G and `false` otherwise.

`LMGNormalClosure(G, H)`

Given a matrix group G defined over a finite field, and a subgroup H of G, the intrinsic returns the normal closure of H in G.

`LMGDerivedGroup(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the derived subgroup of G.

`LMGCommutatorSubgroup(G, H)`

Let g and H be subgroups of $\mathrm{GL}(n, q)$. This intrinsic returns the commutator subgroup of G and H as a subgroup of $\mathrm{GL}(n, q)$.

`LMGIsSoluble(G)`
`LMGIsSolvable(G)`

Given a matrix group G defined over a finite field, the intrinsic returns `true` if G is soluble and `false` otherwise.

`LMGIsNilpotent(G)`

Given a matrix group G defined over a finite field, the intrinsic returns `true` if G is nilpotent and `false` otherwise.

`LMGCompositionSeries(G)`

Given a matrix group G defined over a finite field, the intrinsic returns a composition series for G.

`LMGCompositionFactors(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the composition factors of G. The Handbook entry for `CompositionFactors(G)` of a finite group gives a detailed description of how to interpret the returned sequence.

`LMGChiefSeries(G)`

Given a matrix group G defined over a finite field, the intrinsic returns a chief series for G.

`LMGChiefFactors(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the chief factors G. The Handbook entry for `ChiefFactors(G)` of a finite group gives a detailed description of how to interpret the returned sequence.

`LMGUnipotentRadical(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the unipotent radical U of the matrix group G. A group P of type `GrpPC` and an isomorphism $U \to P$ are also returned.

`LMGSolubleRadical(G)`
`LMGSolvableRadical(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the soluble radical S of G. A group P of type `GrpPC` and an isomorphism $S \to P$ are also returned.

`LMGFittingSubgroup(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the Fitting subgroup S of G. A group P of type `GrpPC` and an isomorphism $S \to P$ are also returned.

`LMGCentre(G)`
`LMGCenter(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the centre of G.

`LMGSylow(G,p)`

Given a matrix group G defined over a finite field, the intrinsic returns a Sylow p-subgroup of G.

`LMGSocleStar(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the inverse image in G of the socle of G/S, where S is the soluble radical of G.

`LMGSocleStarFactors(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the simple direct factors of `LMGSocleStar(G)`/`LMGSolubleRadical(G)`, which may be represented projectively for large classical groups. A list of maps from the factors to G is also returned.

`LMGSocleStarAction(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the map ϕ representing the conjugation action of G on the simple direct factors of `LMGSocleStar(G)`/`LMGSolubleRadical(G)`. The image and kernel of ϕ are also returned.

`LMGSocleStarActionKernel(G)`

Given a matrix group G defined over a finite field, this intrinsic returns three values. The first is the kernel of the conjugation action of G on the simple direct factors of `LMGSocleStar(G)`/`LMGSolubleRadical(G)`. A group P of type `GrpPC` isomorphic to `LMGSocleStarActionKernel(G)`/`LMGSocleStar(G)` and the epimorphism $G \to P$ are also returned.

`LMGSocleStarQuotient(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the quotient group `G`/`LMGSocleStar(G)` represented as a permutation group, with associated epimorphism and kernel.

Example H60E13

We apply the LMG functions to a maximal subgroup of $\mathrm{SL}(12, 5)$.

```
> SetVerbose("LMG", 1);
> C := ClassicalMaximals("L", 12, 5);
> G := C[4];
> LMGFactoredOrder(G);
RandomSchreierBound is 40000
Using CompositionTree on this group
Composition tree computed
Composition series has length 40
Order of group is:  27845944957511377275508129969239234924316406250000000000000\
0000000000000000000
[ <2, 32>, <3, 7>, <5, 66>, <7, 1>, <11, 1>, <13, 3>, <31, 3>, <71, 1>, <313,
1>, <19531, 1> ]
> LMGChiefFactors(G);
Classifying composition factors
Defined PCGroup of solvable radical
Computed PCGroup of SocleKernel/SocleStar
    G
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  A(3, 5)                = L(4, 5)
    *
    |  A(7, 5)                = L(8, 5)
    *
```

```
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(5) (32 copies)
    1
> D := LMGDerivedGroup(G);
RandomSchreierBound is 40000
Using CompositionTree on this group
Composition tree computed
Composition series has length 38
Order of group is:  6961486239377844318877032492309808731079101562500000000000\
0000000000000000000
> LMGIndex(G, D);
4
> SetVerbose("LMG", 0);
> LMGEqual( LMGDerivedGroup(D), D );
true
> S := LMGSolubleRadical(G);
> LMGFactoredOrder(S);
[ <2, 4>, <5, 32> ]
> LMGIsSoluble(G);
false
> LMGIsSoluble(S);
true
> LMGIsNilpotent(S);
false
> #LMGCentre(G);
4
> #LMGCentre(S);
4
```

We carelessly used the standard Magma Order function in the above two commands, but it did not matter, because it was small. We will be more careful next time!

```
> F := LMGFittingSubgroup(G);
> LMGFactoredOrder( LMGCentre(F) );
[ <2, 2>, <5, 32> ]
> P := LMGSylow(G, 5);
> LMGFactoredOrder(P);
[ <5, 66> ]
> LMGEqual( D, LMGNormalClosure(G,P) );
true
> facs, maps := LMGSocleStarFactors(G);
> #facs;
```

```
2
> LMGChiefFactors(facs[1]);
    G
    |  A(7, 5)                = L(8, 5)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
```

Note that, for large classical groups, the socle-star factors are represented projectively.

```
> I := sub< Generic(G) | [ facs[2].i @ maps[2] : i in [1..Ngens(facs[2])] ] >;
> LMGChiefFactors( LMGNormalClosure(G, I) );
    G
    |  A(3, 5)                = L(4, 5)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    1
```

The remaining functions in this section will work only if a permutation representation can be computed for G/L, where L is the soluble radical of G. Apart from `LMGRadicalQuotient` itself, they all operate by solving the problem first in G/L and then lifting the solution through elementary abelian layers of L. Results are returned using the same formats as for other types of finite groups.

`LMGRadicalQuotient(G)`

Given a matrix group G defined over a finite field, the intrinsic returns a permutation group P isomorphic to G/L, where L is the soluble radical of G. An epimorphism $G \to P$ and its kernel L are also returned.

Of course, this will only work if G/L has such a representation of sufficiently small degree. This function is called implicitly as a first step in all of the remaining functions in this section.

`LMGCentraliser(G, g)`

`LMGCentralizer(G, g)`

Given a matrix group G defined over a finite field, the intrinsic returns the centraliser in the matrix group G of $g \in G$.

`LMGIsConjugate(G, g, h)`

Given a matrix group G defined over a finite field, the intrinsic returns `true` if the elements g, h in G are conjugate. If so, a conjugating element will also be returned.

`LMGClasses(G)`

`LMGConjugacyClasses(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the conjugacy classes of G.

`LMGNormaliser(G, H)`

`LMGNormalizer(G, H)`

Given a matrix group G defined over a finite field, and a subgroup H of G, the intrinsic returns the normaliser of H in G.

`LMGIsConjugate(G, H, K)`

Given a matrix group G defined over a finite field, and subgroups H and K of G, the intrinsic returns `true` if the subgroups H and K are conjugate in G. If so, a conjugating element will also be returned.

`LMGMaximalSubgroups(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the maximal subgroups of G.

60.8 Unipotent Matrix Groups

The *power-conjugate presentation* is a very efficient way of representing a unipotent group; see Chapter 63 for more information. In this section we describe a number of functions for finding such a *PC-presentation* for a unipotent matrix group defined over a finite field.

The algorithm used is a straightforward echelonisation-like procedure.

`UnipotentMatrixGroup(G)`

Given a matrix group G defined over a finite field, the intrinsic constructs a known unipotent matrix group from G. Note that MAGMA does not at this stage check that G is in fact unipotent.

`WordMap(G)`

Given a unipotent matrix group G defined over a finite field, the intrinsic constructs the *word map* for G. The word map is a map from G to the group of straight-line programs on n generators, where n is the number of generators of G. More information on SLP-groups may be found in Chapter 76.

Example H60E14

We construct a unipotent matrix group, and use the word map.

```
> G := MatrixGroup<4, GF(5) | [1,1,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1],
>       [1,-1,0,0, 0,1,1,0, 0,0,1,0, 0,0,0,1]>;
> G;
MatrixGroup(4, GF(5))
Generators:
    [1 1 0 0]
    [0 1 0 0]
    [0 0 1 0]
    [0 0 0 1]

    [1 4 0 0]
    [0 1 1 0]
    [0 0 1 0]
    [0 0 0 1]
> IsUnipotent(G);
true
>
> G := UnipotentMatrixGroup(G);
> g := GL(4,5)![1,4,4,0, 0,1,3,0, 0,0,1,0, 0,0,0,1];
> g in G;
true
> phi := WordMap(G);
> phi;
Mapping from: GL(4, GF(5)) to SLPGroup(2) given by a rule [no inverse]
>
> assert g in G;
```

```
> wg := phi(g); wg;
function(G)
    w6 := G.1^4; w1 := G.1^-4; w2 := G.2 * w1; w7 := w2^3; w8 := w6 *
    w7; w3 := G.1^-1; w4 := G.1^w2; w5 := w3 * w4; w9 := w5^2; w10 :=
    w8 * w9; return w10;
end function
> Evaluate(wg, G);
[1 4 4 0]
[0 1 3 0]
[0 0 1 0]
[0 0 0 1]
> Evaluate(wg, G) eq g;
true
```

`PCPresentation(G)`

Given a unipotent matrix group G defined over a finite field, the intrinsic constructs a PC-presentation for G. It returns a finite soluble group H as first return value, a map from G to H as the second value, and a map from H to G as the third.

`Order(G)`
`#G`
`FactoredOrder(G)`

Given a unipotent matrix group G defined over a finite field, this intrinsic returns the order of G as an integer or as a factored integer (depending upon the choice of intrinsic). It is faster than the standard matrix group order intrinsic because of the use of the PC-presentation of G.

`g in G`

Given a matrix g and a unipotent matrix group G defined over a finite field, the intrinsic returns `true` if g is an element of G, and `false` otherwise. It is faster than the standard matrix group membership intrinsic because of the use of the PC-presentation of G.

Example H60E15

We construct the PC-presentation of some Sylow subgroup and demonstrate the use of the `FactoredOrder` function.

```
> G := UnipotentMatrixGroup(ClassicalSylow(GL(9,7), 7));
> H,phi,psi := PCPresentation(G);
> phi;
Mapping from: GrpMatUnip: G to GrpPC: H given by a rule [no inverse]
> psi;
Mapping from: GrpPC: H to GrpMatUnip: G
> phi(G.2);
```

```
H.9
> psi(H.3);
[1 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0]
[0 0 1 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 1]
> FactoredOrder(G);
[ <7, 36> ]
```

60.9 Bibliography

[**AMPS10**] S. Ambrose, S.H̃. Murray, C.Ẽ. Praeger, and C. Schneider. Constructive membership testing in black-box classical groups. In *Proceedings of The Third International Congress on Mathematical Software*, number 6327 in Lecture Notes in Computer Science, pages 54–57, Basel, 2010. Springer.

[**Asc84**] M. Aschbacher. On the maximal subgroups of the finite classical groups. *Invent. Math*, 76:469–514, 1984.

[**BHLGO11**] H. Bäärnhielm, Derek Holt, C.R. Leedham-Green, and E.A. O'Brien. A practical model for computation with matrix groups. *preprint*, 2011.

[**Bra00**] J.N. Bray. An improved method of finding the centralizer of an involution. *Arch. Math. (Basel)*, 74(1):241–245, 2000.

[**Cos09**] E. Costi. *Constructive membership testing in classical groups.* PhD thesis, Queen Mary, University of London, 2009.

[**DLLGO13**] Heiko Dietrich, Frank Lübeck, C.R. Leedham-Green, and E.A. O'Brien. Constructive recognition of classical groups in even characteristic. *J. Algebra*, 2013.

[**GH97**] S.P. Glasby and R.B. Howlett. Writing representations over minimal fields. *Comm. Algebra*, 25(6):1703–1711, 1997.

[**GLGO05**] S.P. Glasby, C.R. Leedham-Green, and E.A. O'Brien. Writing projective representations over subfields. *J. Algebra*, 295:51–61, 2005.

[**HLGOR96a**] Derek F. Holt, C.R. Leedham-Green, E.A. O'Brien, and Sarah Rees. Computing decompositions for modules with respect to a normal subgroup. *J. Algebra*, 184:818–838, 1996.

[**HLGOR96b**] Derek F. Holt, C.R. Leedham-Green, E.A. O'Brien, and Sarah Rees. Testing matrix groups for primitivity. *J. Algebra*, 184:795–817, 1996.

[**LG01**] Charles R. Leedham-Green. The computational matrix group project. In *Groups and computation, III (Columbus, OH, 1999)*, volume 8 of *Ohio State Univ. Math. Res. Inst. Publ.*, pages 229–247. de Gruyter, Berlin, 2001.

[**LGO**] C.R. Leedham-Green and E.A. O'Brien. Short presentations for classical groups. *preprint.*

[**LGO97a**] C.R. Leedham-Green and E.A. O'Brien. Recognising tensor products of matrix groups. *Internat. J. Algebra Comput.*, 7:541–559, 1997.

[**LGO97b**] C.R. Leedham-Green and E.A. O'Brien. Tensor Products are Projective Geometries. *J. Algebra*, 189:514–528, 1997.

[**LGO02**] C.R. Leedham-Green and E.A. O'Brien. Recognising tensor-induced matrix groups. *J. Algebra*, 253:14–30, 2002.

[**LGO09**] C.R. Leedham-Green and E.A. O'Brien. Constructive recognition of classical groups in odd characteristic. *J. Algebra*, 322:833–881, 2009.

[**Nie05**] Alice C. Niemeyer. Constructive recognition of normalisers of small extraspecial matrix groups. *Internat. J. Algebra Comput.*, 15:367–394, 2005.

[**NS06**] Max Neunhöffer and Ákos Seress. A data structure for a uniform approach to computations with finite groups. In *ISSAC 2006*, pages 254–261. ACM, New York, 2006.

[**O'B06**] E.A. O'Brien. Towards effective algorithms for linear groups. In *Finite Geometries, Groups and Computation*, pages 163–190. De Gruyer, 2006.

[**O'B11**] E.A. O'Brien. Algorithms for matrix groups. In *Groups St Andrews (Bath)*, volume 388 of *LMS Lecture Notes*, pages 297–323. Cambridge University Press, 2011.

61 MATRIX GROUPS OVER INFINITE FIELDS

Chapter 61
MATRIX GROUPS OVER INFINITE FIELDS

61.1 Overview

In this chapter we provide algorithms for computing with a group G given by a finite set $S = \{g_1, \ldots, g_r\}$ of invertible $n \times n$ matrices over an infinite field K. The algorithms are based on special techniques developed for computing in this class of groups ([DF08, DF09, DFO09, DEF09, DFO12, DFO11]), which rely on properties of finitely generated linear groups.

The group G is defined over the subring R of K generated by the entries of the matrices $g_i, {g_i}^{-1}$, $1 \leq i \leq r$. If ρ is an ideal of R, then it induces a congruence homomorphism from $GL(n, R)$ onto $GL(n, R/\rho)$, which replaces every entry of an element in S by its image in R/ρ. Our techniques depend on the construction of a congruence homomorphism with the property that all torsion elements of its kernel G_ρ (called a congruence subgroup) are unipotent. The existence of a normal subgroup of finite index in G with such a property was proved by Selberg and Wehrfritz. One advantage of the congruence homomorphism techniques is that they replace the ground domain by a domain that is more convenient for computing. In particular, if the ideal ρ is maximal, then we get a reduction to a finite field R/ρ. For more details on the method see [DF08, Section 3].

In this chapter we provide three sets of functions based on the above techniques.

(a) Functions which test finiteness of matrix groups over a wide range of infinite domains. These functions are an implementation of algorithms developed in [DF09, DFO09, DFO12]. Together with other currently available algorithms for deciding finiteness, they enable testing finiteness of a finitely generated linear group over an arbitrary field (subject to special representation of input data). Additionally, if a group is found to be finite, then we can construct an isomorphic copy over a finite field, and use that for further structural investigation of the group.

(b) Functions for testing various properties of infinite matrix groups. These functions test whether G is soluble-by-finite or soluble, nilpotent-by-finite or nilpotent, abelian-by-finite, or central-by-finite. In effect, they provide access to the first publicly available implementations of algorithms to decide the "Tits alternative" for a linear group. If G is soluble-by-finite we can test whether it is completely reducible. These functions are an implementation of algorithms developed in [DFO11].

(c) Functions for testing nilpotency and computing with nilpotent matrix groups. These functions are an implementation of algorithms developed in [DF08], which in turn are based on an implementation of algorithms in [DF06] for computing with nilpotent matrix groups over finite fields. The functions may also be used for investigating the structure of nilpotent matrix groups. In particular, special algorithms have been developed for deciding finiteness of nilpotent matrix groups. Functions are also available to decide irreducibility and primitivity for finite nilpotent matrix groups over number fields and function fields

in zero characteristic; these algorithms, developed and implemented by Tobias Rossmann, are described in [Ros10, Ros11].

Since G is finitely generated, it is defined over a finitely generated subfield of K. Hence, the main fields to be considered are finite degree extensions of $F(x_1, \ldots, x_m)$, where the $x_1, \ldots, x_m$ are algebraically independent indeterminates, $m \geq 0$, and the coefficient field F is an number field or a finite field.

For a recent survey of work in the area of computing with matrix groups over infinite fields, we refer to [DEF09].

Verbose output for these functions can be obtained with `SetVerbose ("Infinite", 1);`

61.2 Construction of Congruence Homomorphisms

In this section, K is a finite degree extension of $F(x_1, \ldots, x_m)$, where F is Q, a number field, or a finite field. Also $m \geq 0$ if $charF = 0$, and $m > 0$ otherwise.

`CongruenceImage(G : parameters)`

`Virtual`	BoolElt	*Default :* `false`
`Prime`	RngIntElt	*Default :* 3
`Limit`	RngIntElt	*Default :* 10
`ExtDegree`	RngIntElt	*Default :* 1

If G is a finitely generated subgroup of $\mathrm{GL}(n, K)$, then G has a normal subgroup N whose torsion elements are unipotent; so N is torsion-free if K has characteristic 0.

This function constructs a *congruence homomorphism* from G into $\mathrm{GL}(n, \mathbf{F}_q)$ for some prime power q; its kernel is N. If $charK$ is positive, then $\mathbf{F}_q$ has the same characteristic.

For a detailed description of the congruence homomorphisms see [DFO12, Section 3]. The function returns the congruence image H, the congruence homomorphism, and the list of images of generators of G.

If the optional parameter `Virtual` is set to `true` then the congruence homomorphism satisfies additional properties [DFO11]. In particular it can be used to test whether G satisfies the "virtual" properties described in Section 61.4.

The optional parameter `Prime` applies if K has characteristic 0: if `Prime` is positive, then it is a lower bound for the characteristic of the congruence image; if it is 0 then the function returns a congruence image defined over a field of characteristic 0.

The optional parameter `Limit` applies to groups defined over (rational) function fields. If $charK > 0$, then we consider extensions of F to degree `Limit` only; otherwise we examine tuples in the ring of integers mod `Limit`.

The optional parameter `ExtDegree` applies to groups defined over (algebraic) function fields of positive characteristic: we construct a congruence image over an extension of (at least) this degree of coefficient field.

61.3 Testing Finiteness

In this section, K is a finite degree extension of the field $F(x_1, \ldots, x_m)$, where F is Q, a number field, or a finite field. Also $m \geq 0$ if $charF = 0$, and $m > 0$ otherwise.

`IsFinite(G : parameters)`

`NumberRandom`	RngIntElt	*Default :* 10
`Presentation`	MonStgElt	*Default :* "*CT*"
`Small`	RngIntElt	*Default :* 10^5
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Algebra`	BoolElt	*Default :* `true`
`Nilpotent`	BoolElt	*Default :* `false`
`UseCongruence`	BoolElt	*Default :* `false`
`DetermineOrder`	BoolElt	*Default :* `false`
`Prime`	RngIntElt	*Default :* 3

Let G be a finitely generated subgroup of $\mathrm{GL}(n, K)$. If G is finite then the function returns `true`, otherwise `false`. The function is an implementation of algorithms from [DFO12, DF09, DFO09, DF08].

The algorithm first tests whether `NumberRandom` random elements of G have finite order.

If the optional parameter `Algebra` is `true` and K is a function field of characteristic zero (resp. positive characteristic), then we use the "algebra algorithm" of [DF09] (resp. [DFO09]) to decide finiteness.

Otherwise, we prove that G is finite by first constructing a congruence homomorphism, then a presentation for the congruence image, and finally evaluates its relations to obtain normal generators for the congruence kernel. If $charK = 0$, then the kernel should be trivial, otherwise the kernel is unipotent.

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

If K is Q or a number field and `UseCongruence` is `true`, then use congruence homomorphism machinery to decide; otherwise use default algorithm.

If G is known to be nilpotent then by setting the optional parameter `Nilpotent` to `true`, the function will call a special procedure for testing finiteness of nilpotent groups (see [DF08, Section 4.3]).

If the optional parameter `DetermineOrder` is set to `true`, and G is finite, then the function returns the order of G. This may sometimes be more expensive than deciding finiteness.

The optional parameter `Prime` applies if K has characteristic 0: if `Prime` is positive, then it is a lower bound for the characteristic of the congruence image; if it is 0 then the function constructs a congruence image defined over a field of characteristic 0.

`IsomorphicCopy(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`Small`	RngIntElt	*Default :* 10^5
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Verify`	BoolElt	*Default :* `false`
`Algebra`	BoolElt	*Default :* `false`
`StartDegree`	RngIntElt	*Default :* 1
`EndDegree`	RngIntElt	*Default :* 5
`CompletelyReducible`	BoolElt	*Default :* `false`

The input is a finite subgroup G of $\mathrm{GL}(n, K)$. If the function succeeds, then it returns `true` and an isomorphic copy of G in $\mathrm{GL}(n, \mathbf{F}_q)$ where q is a prime power; otherwise it returns `false`. A description of the method used is in [DFO12, Section 4.3]. If $charK$ is positive, then $\mathbf{F}_q$ has the same characteristic. Note that the function always succeeds if K has zero characteristic.

If the optional parameter `Algebra` is `true` and K is a function field of characteristic zero (resp. positive characteristic), then we use the "algebra algorithm" of [DF09] (resp. [DFO09]) to construct an isomorphic copy.

Otherwise we prove that a congruence homomorphism is an isomorphism by constructing a presentation for the congruence image and evaluating its relations to obtain normal generators for the congruence kernel.

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

If the optional parameter `Verify` is set to `true` then we first check whether G is finite.

If the characteristic of the coefficient field F is positive, then we investigate extensions of F in the range `StartDegree` ... `EndDegree`.

If the optional parameter `CompletelyReducible` is set to `true` then we use a more efficient algorithm to construct the isomorphic copy.

`Order(G : parameters)`

`Verify`	BoolElt	*Default :* `false`
`UseCongruence`	BoolElt	*Default :* `false`

Given a finite subgroup G of $\mathrm{GL}(n, K)$, the function returns the order of G by applying `IsomorphicCopy` to G.

If the optional parameter `Verify` is set to `true`, then we first check that G is finite.

If K is Q or a number field and `UseCongruence` is `true`, then use congruence homomorphism machinery to decide; otherwise use default algorithm.

61.4 Deciding Virtual Properties of Linear Groups

In this section, K is a finite degree extension of $F(x_1, \ldots, x_m)$, where F is Q, a number field, or a finite field. Also $m \geq 0$ if $charF = 0$, and $m > 0$ otherwise.

We describe algorithms to decide various "virtual" properties of a finitely generated linear group over an infinite field. Details of the algorithms can be found in [DFO11].

`IsSolubleByFinite(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5

This function takes as input a finitely generated matrix group G over K, and tests whether G is soluble-by-finite. If so, it returns `true`, otherwise `false`. Note that currently the function is valid only for $p > n$ if K has characteristic $p > 0$.

The algorithm first constructs a congruence homomorphism, then a presentation for the congruence image, and finally evaluates its relations to obtain normal generators for the congruence kernel. For further details, see [DFO11, Section 3.2].

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

IsPolycyclicByFinite(G : *parameters*)

Presentation	MonStgElt	*Default :* "*CT*"
OrderLimit	RngIntElt	*Default :* 10^{12}
Small	RngIntElt	*Default :* 10^5

This function takes as input a finitely generated matrix group G over Z, and tests whether G is polycyclic-by-finite. If so, it returns `true`, otherwise `false`. See [DFO11, Section 3.2] for details.

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

IsNilpotentByFinite(G : *parameters*)

Presentation	MonStgElt	*Default :* "*CT*"
OrderLimit	RngIntElt	*Default :* 10^{12}
Small	RngIntElt	*Default :* 10^5

This function takes as input a finitely generated matrix group G over K, and tests whether G is nilpotent-by-finite. If so, it returns `true`, otherwise `false`. Here K must currently be Q, a number field, or an (algebraic) function field with a single indeterminate.

The algorithm first constructs a congruence homomorphism, then a presentation for the congruence image, and finally evaluates its relations to obtain normal generators for the congruence kernel. Further details of the algorithm can be found in [DFO11, Section 5.2].

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

`IsAbelianByFinite(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5

This function takes as input a finitely generated matrix group G over K, and tests whether G is abelian-by-finite. If so, it returns `true`, otherwise `false`. As before, K must currently be Q, a number field, or an (algebraic) function field with a single indeterminate.

The algorithm first constructs a congruence homomorphism, then a presentation for the congruence image, and finally evaluates its relations to obtain normal generators for the congruence kernel. Further details of the algorithm can be found in [DFO11, Section 5.2].

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

`IsCentralByFinite(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5
`CompletelyReducible`	BoolElt	*Default :* `false`

This function takes as input a finitely generated matrix group G over a field K, and tests whether G is central-by-finite. If so, it returns `true`, otherwise `false`. Here K is (a finite degree extension of) $F(x_1, \ldots, x_m)$, where F is Q or a number field.

The algorithm first constructs a congruence homomorphism, then a presentation for the congruence image, and finally evaluates its relations to obtain normal generators for the congruence kernel. Further details of the algorithm can be found in [DFO11, Section 5.3].

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional

argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

If the optional parameter `CompletelyReducible` is set to `true` then we use a more efficient algorithm to test whether G is central-by-finite.

61.5 Other Properties of Linear Groups

In this section, K is a finite degree extension of $F(x_1, \ldots, x_m)$, where F is Q, a number field, or a finite field, and $m \geq 0$.

`IsCompletelyReducible(G : parameters)`

`SolubleByFinite`	BoolElt	*Default :* `false`
`NilpotentByFinite`	BoolElt	*Default :* `false`
`AbelianByFinite`	BoolElt	*Default :* `false`
`Nilpotent`	BoolElt	*Default :* `false`
`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5

This function takes as input a finitely generated matrix group G over K, and tests whether G is completely reducible. If so, it returns `true`, otherwise `false`.

The algorithm used is described in [DFO11, Section 4]. It applies only if G is soluble-by-finite, nilpotent-by-finite, or abelian-by-finite. Hence one (and only one) of the four optional arguments `SolubleByFinite, NilpotentByFinite, AbelianByFinite, Nilpotent` must be true. In particular, if `Nilpotent` is set to be true, then a more efficient algorithm (from [DF08]) is used.

In positive characteristic p, if p divides the order of the congruence image of G then currently the algorithm cannot decide complete reducibility of G.

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

`IsUnipotent(G)`

This function takes as input a finitely generated matrix group G defined over an exact field F, and tests whether G is unipotent, i.e., whether it is conjugate in $\mathrm{GL}(n, F)$ to a group of upper unitriangular matrices. If G is unipotent then the function returns `true` and a change-of-basis matrix $c \in \mathrm{GL}(n, F)$ such that G^c is upper unitriangular, otherwise `false`. See [DF06, Section 2.1] for details of the algorithm.

`IsNilpotent(G)`

Let G be a finitely generated subgroup of $\mathrm{GL}(n, K)$. This function returns `true` if G is nilpotent; otherwise it returns `false`. If K is finite then the function is an implementation of the algorithm of [DF06]. If K is infinite then the function is similar to the algorithm in [DF08], and is based on the construction of a homomorphic image H of G via `CongruenceImage`.

`IsSoluble(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5
`UseCongruence`	BoolElt	*Default :* `false`

Let G be a finitely generated subgroup of $\mathrm{GL}(n, K)$. This function returns `true` if G is soluble; otherwise it returns `false`. If K is infinite and has characteristic $p > 0$, then the algorithm is applicable only for $p > n$. For details see [DFO11, Section 3.2].

If K is Q or a number field and `UseCongruence` is `true`, then use congruence homomorphism machinery to decide; otherwise use default algorithm.

The other optional arguments are those described above for `IsSolubleByFinite`.

`IsPolycyclic(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5

This function takes as input a finite matrix group G over Z, and tests whether G is polycyclic. If so, it returns `true`, otherwise `false`.

The optional arguments are those described above for `IsSolubleByFinite`.

`HasFiniteOrder (g : parameters)`

`UseCongruence`	BoolElt	*Default :* `false`

Let g be an invertible matrix defined over Z, Q, a number field, a function field, or an algebraic function field.

If g has finite order, then return `true` and, if known, a multiplicative upper bound for the order of g; else return `false`.

If g is defined over Z, Q, or a number field and `UseCongruence` is `true`, then use congruence homomorphism machinery to decide; otherwise use default algorithm.

61.6 Other Functions for Nilpotent Matrix Groups

`SylowSystem(G : parameters)`

`Verify` BoolElt *Default :* `false`

Given a nilpotent matrix group G over a finite field, this function constructs one Sylow p-subgroup for each prime p dividing $|G|$ using the algorithm of [DF06]. If the optional parameter `Verify` is set to `true`, then we first verify that G is nilpotent.

The next two functions were developed and implemented by Tobias Rossmann.

`IsIrreducibleFiniteNilpotent(G : parameters)`

`DecideOnly` BoolElt *Default :* `false`

`Verify` BoolElt *Default :* `false`

Let G be a finite nilpotent matrix group over K, where K is a number field or a rational function field over a number field. The function returns `true` if G is irreducible or `false` and a proper submodule of `GModule(G)`. The construction of a submodule can be suppressed by setting `DecideOnly` to `true`. If the optional parameter `Verify` is set to `true`, then the function checks if G is nilpotent and finite. The algorithm used for irreducibility testing is described in [Ros10a].

`IsPrimitiveFiniteNilpotent(G : parameters)`

`DecideOnly` BoolElt *Default :* `false`

`Verify` BoolElt *Default :* `false`

Let G be an irreducible finite nilpotent matrix group over K, where K is a number field or a rational function field over a number field. The function returns `true` if G is primitive, or `false` and a system of imprimitivity for G given as a sequence of subspaces of `RSpace(G)`. The construction of a system of imprimitivity can be suppressed by setting `DecideOnly` to `true`. If the optional parameter `Verify` is set to `true`, then the function checks if G is nilpotent and finite. The algorithm used for primitivity testing is described in [Ros10b].

61.7 Examples

Example H61E1

```
> Q := Rationals ();
> F<t>:= RationalFunctionField (Q);
> M:= MatrixAlgebra (F, 3);
> a:= M![-1, 2*t^2, -2*t^4 - 2*t^3 - 2*t^2, 0, 1, 0, 0, 0, 1];
> b:= M![1, 0, 0, 1/t^2, -1, (2*t^3 - 1)/(t - 1), 0, 0, 1];
> c:= M![t, -t^3 + t^2, t^5 - t^2 - t, t^2, -t^4, (t^8 - t^5 + 1)/
> (t^2 - t), (t - 1)/t, -t^2 + t, t^4 - t];
> G:= sub<GL(3,F)|a,b,c>;
> IsFinite(G);
```

```
true
> flag, H := IsomorphicCopy(G);
> H;
MatrixGroup(3, GF(3))
Generators:
    [2 2 1]
    [0 1 0]
    [0 0 1]

    [1 0 0]
    [1 2 0]
    [0 0 1]

    [2 2 2]
    [1 2 0]
    [2 1 2]
> #H;
48
```

Example H61E2

```
> F<t>:= RationalFunctionField (GF(5));
> M:= MatrixAlgebra (F, 6);
> a:= M![2, 2*t^2, 4, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0,
> 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1];
> b:= M![(4*t + 4)/t, 4*t, (t + 1)/t, 0, t, t^2 + t, 0, 4, 0, 0, 0,
> 1/t, 4/t, t^2 + 4*t, 1/t, 0, 0, 0, 0, 4*t, 0, 0, 0, 0, 0, 0, 4, 4,
> 0, 0, 0, 0, 0, 4, 0, 0];
> G:= sub<GL(6,F)|a,b>;
> IsFinite(G);
true
> flag, H := IsomorphicCopy (G);
> flag;
true
> H;
MatrixGroup(6, GF(5)) of order 2^7 * 3 * 5^4 * 31
Generators:
    [2 2 4 1 0 0]
    [0 2 0 0 0 0]
    [0 0 1 1 0 0]
    [0 0 0 1 0 0]
    [0 0 0 0 1 1]
    [0 0 0 0 0 1]

    [3 4 2 0 1 2]
    [0 4 0 0 0 1]
    [4 0 1 0 0 0]
```

```
    [0 4 0 0 0 0]
    [0 0 4 4 0 0]
    [0 0 0 4 0 0]
> #H;
7440000
```

Example H61E3

```
>  L<t> := RationalFunctionField (GF (5^2));
>  G := GL (2, L);
>  a := G![t,1,0,-1];
>  b:= G![t/(t + 1), 1, 0, 1/t];
>  H := sub <GL(2, L) | a, b>;
> f :=IsFinite(H);
> f;
false
> IsSolubleByFinite (H);
true
> IsCompletelyReducible (H);
false
```

Example H61E4

```
> G := MatrixGroup<3, IntegerRing() |
>  [ 5608, 711, -711, 6048, 766, -765, 1071, 135, -134 ],
>  [ 1, -2415, 5475, 0, 4471, -10140, 0, 780, -1769 ],
>  [ 5743, -5742, 639, -576, 577, -72, -711, 711, -80 ],
>  [ 526168, -618507, 729315, 621984, -731138, 862125,
>        274455, -322620, 380419      ]  ,
>  [ 648226, -4621455, 9226791, 660687, -4710305, 9404184,
>        85626, -610473, 1218820 ],
>  [ 32581, -39465, 46350, 53100, -64319, 75540, 24210,
>        -29325, 34441 ]>;
> IsFinite (G);
false
> IsSolubleByFinite (G);
false
> IsNilpotentByFinite (G);
false
> time IsCentralByFinite (G);
false
> IsAbelianByFinite (G);
false
```

Example H61E5

```
> Q<z> := QuadraticField(5);
> O<w> := sub< MaximalOrder(Q) | 7 >;
> G := GL(2, Q);
> x := G![1,1+w,0,w];
> y :=  G![-1/2, 2, 2 + w, 5 + w^2];
> H:=sub<G | x, y>;
> IsFinite (H);
false
> IsSolubleByFinite (H);
false
```

Example H61E6

```
>  R<x> := PolynomialRing(Integers());
>  K<y> := NumberField(x^4-420*x^2+40000);
>  G := GL (2, K);
>  a := G![y,1,0,-1];
>  b:= G![y/(y + 1), 1, 0, 1/y];
>  H := sub <GL(2, K) | a, b>;
> time IsFinite(H);
false
```

Example H61E7

```
> /* example over algebraic extension of a function field */
>
> R<u> := FunctionField (Rationals ());
> v := u; w := -2 * v;
> Px<X> := PolynomialRing (R);
> Py<Y> := PolynomialRing (R);
> f :=  Y^2- 3 * u * X * Y^2  + v * X^3;
> facs := Factorisation (f);
> F:=ext <R | facs[2][1]>;
> F;
Algebraic function field defined over Univariate rational function
field over Rational Field by Y - 1/2/u
>
> n := 3;
> G:= GL(n,F);
> Z := 4 * X * Y;
> MA:= MatrixAlgebra(F,n);
> h1:= Id(MA);
> h1[n][n]:= (X^2+Y+Z+1);
> h1[1][n]:= X+1;
```

```
> h1[1][n]:= X+1;
> h1[1][1]:=(Z^5-X^2*Z+Z*X*Y);
> h1[2][1]:=1-X*Y*Z;
> h1[2][n]:= X^20+X*Y^15+Y^10+Z^4*Y*X^5+1;
> h2:= Id(MA);
> h2[n][n]:= (X^7+Z^6+1);
> h2[1][n]:= X^2+X+1;
> h2[1][1]:=(Y^3+X^2+X+1);
> h2[1][1]:=(Y^3+X^2+X+1);
> h2[2][1]:=1-X^2;
> h2[2][n]:= X^50+Y^35+X^20+X^13+Y^2+1;
> G := sub< GL(n, F) | h1, h2>;
> G;
MatrixGroup(3, F)
Generators:
    [1/u^10 0 (u + 1/2)/u]
    [(u^4 - 1/4)/u^4 1 (u^20 + 1/1024*u^10 + 1/64*u^6 + 1/65536*u^4 +
        1/1048576)/u^20]
    [0 0 (u^2 + 1/2*u + 5/4)/u^2]

    [(u^3 + 1/2*u^2 + 1/4*u + 1/8)/u^3 0 (u^2 + 1/2*u + 1/4)/u^2]
    [(u^2 - 1/4)/u^2 1 (u^50 + 1/4*u^48 + 1/8192*u^37 + 1/1048576*u^30 +
        1/34359738368*u^15 + 1/1125899906842624)/u^50]
    [0 0 (u^12 + 1/128*u^5 + 1)/u^12]
> time IsFinite(G);
false
Time: 0.010
> time IsSolubleByFinite (G);
true
```

Example H61E8

```
> F := GF(2);
> P := PolynomialRing (F);
> P<t> := PolynomialRing (F);
> F := ext < F | t^2+t+1>;
> G := GL (2, FunctionField (F));
> a := G![1,1/t, 0, 1];
> b := [1,1/(t + 1), 0, 1];
> c := [1,1/(t^2 + t + 1), 0, 1];
> d := [1,1/(t^2 + t), 0, 1];
> G := sub < G | a,b,c,d>;
> time IsFinite (G);
true
> f, I, tau := IsomorphicCopy (G);
> f;
```

```
true
```

Example H61E9

```
> // irreducible but (evidently) imprimitive
> K<w> := QuadraticField (2);
> G := MatrixGroup< 8, K |
>     [1/2*w,1/2*w,0,0,0,0,0,0,-1/2*w,1/2*w,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
>      0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
>      0,0,0,1],
>     [1,0,0,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,
>      0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1],
>     [0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,
>     0,0,0,0,1,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0] >;
> G;
MatrixGroup(8, K)
Generators:
    [ 1/2*w  1/2*w        0        0        0        0        0        0]
    [-1/2*w  1/2*w        0        0        0        0        0        0]
    [       0        0        1        0        0        0        0        0]
    [       0        0        0        1        0        0        0        0]
    [       0        0        0        0        1        0        0        0]
    [       0        0        0        0        0        1        0        0]
    [       0        0        0        0        0        0        1        0]
    [       0        0        0        0        0        0        0        1]

    [ 1  0  0  0  0  0  0  0]
    [ 0 -1  0  0  0  0  0  0]
    [ 0  0  1  0  0  0  0  0]
    [ 0  0  0  1  0  0  0  0]
    [ 0  0  0  0  1  0  0  0]
    [ 0  0  0  0  0  1  0  0]
    [ 0  0  0  0  0  0  1  0]
    [ 0  0  0  0  0  0  0  1]

    [0 0 1 0 0 0 0 0]
    [0 0 0 1 0 0 0 0]
    [0 0 0 0 1 0 0 0]
    [0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 1 0]
    [0 0 0 0 0 0 0 1]
    [1 0 0 0 0 0 0 0]
    [0 1 0 0 0 0 0 0]
> IsIrreducibleFiniteNilpotent(G);
true
> r, B := IsPrimitiveFiniteNilpotent(G);
> r;
```

```
false
> #B;
2
```

Example H61E10

```
> M:= MatrixAlgebra (GF(17), 4);
> a:= M![5, 5, 3, 3, 0, 5, 0, 3, 16, 16, 14, 14, 0, 16, 0, 14];
> b:= M![9, 9, 0, 0, 0, 9, 0, 0, 10, 10, 8, 8, 0, 10, 0, 8];
> G:= sub<GL(4,17)|a,b>;
> IsNilpotent(G);
true
> SylowSystem (G);
[
    MatrixGroup(4, GF(17))
    Generators:
        [ 5  0  3  0]
        [ 0  5  0  3]
        [16  0 14  0]
        [ 0 16  0 14]

        [ 9  0  0  0]
        [ 0  9  0  0]
        [10  0  8  0]
        [ 0 10  0  8],

    MatrixGroup(4, GF(17))
    Generators:
        [ 1  1  0  0]
        [ 0  1  0  0]
        [ 0  0  1  1]
        [ 0  0  0  1]
]
> Order(G);
8704
```

Example H61E11

```
> R<s>:= QuadraticField(-1);
> F<t>:= FunctionField(R);
> M:= MatrixAlgebra (F, 2);
> a:= M![-s*t^2 + 1, s*t^3, -s*t, s*t^2 + 1];
> b:= M![t^2 - 3*t + 1, 0, 0, t^2 - 3*t + 1];
> G:= sub<GL(2,F)|a,b>;
> IsNilpotent(G);
true
```

```
> IsFinite(G);
false
```

61.8 Bibliography

[**DEF09**] A. S. Detinko, B. Eick, and D. L. Flannery. Computing with matrix groups over infinite fields. *preprint (15pp.)*, 2009.

[**DF06**] A. S. Detinko and D. L. Flannery. Computing in nilpotent matrix groups. *LMS J. Comput. Math.*, 9:104–134 (electronic), 2006.

[**DF08**] A. S. Detinko and D. L. Flannery. Algorithms for computing with nilpotent matrix groups over infinite domains. *J. Symbolic Comput.*, 43:8–26, 2008.

[**DF09**] A. S. Detinko and D. L. Flannery. On deciding finiteness of matrix groups. *J. Symbolic Comput.*, 44:1037–1043, 2009.

[**DFO09**] A. S. Detinko, D. L. Flannery, and E. A. O'Brien. Deciding finiteness of matrix groups in positive characteristic. *J. Algebra*, 322:4151–4160, 2009.

[**DFO11**] A. S. Detinko, D. L. Flannery, and E. A. O'Brien. Algorithms for the Tits alternative and related problems. *J. Algebra*, 344:397–406, 2011.

[**DFO12**] A. S. Detinko, D. L. Flannery, and E. A. O'Brien. Recognition of finite matrix groups over infinite fields. *J. Symbolic Comput.*, 2012.

[**Ros10**] T. Rossmann. Irreducibility testing of finite nilpotent linear groups. *J. Algebra*, 324:1114–1124, 2010.

[**Ros11**] T. Rossmann. Primitivity testing of finite nilpotent linear groups. *LMS JCM*, 14:87–98, 2011.

62 MATRIX GROUPS OVER Q AND Z

Chapter 62
MATRIX GROUPS OVER Q AND Z

62.1 Overview

In addition to the functionality explained in Chapter 61 and the functions that are available for all finite (matrix) groups, MAGMA can also compute normalizers and centralizers of a finite integral matrix group G in $\mathrm{GL}_n(\mathbf{Z})$ as well as decide conjugacy in $\mathrm{GL}_n(\mathbf{Z})$ and $\mathrm{GL}_n(\mathbf{Q})$.

These algorithms are based on the sublattice machinery (see Section 31.3.5) and the enumeration of G-perfect forms. They are explained in [OPS98, Opg01]. The algorithms perform very well, as long as the space of G-invariant symmetric forms has small dimension (say less than 15) and the index of the groups in their Bravais groups is not too large.

The databases of maximal finite irreducible rational, integral, symplectic and quaternionic matrix groups are explained in Chapter 66.

62.2 Invariant Forms

Let G be a finite matrix group $G < \mathrm{GL}_n(\mathbf{Q})$. A matrix $F \in M_n(\mathbf{Q})$ is G-invariant if $gFg^{tr} = F$ for all $g \in G$.

`PositiveDefiniteForm(G)`

For a finite integral or rational matrix group G, return a positive definite symmetric G-invariant form.

`InvariantForms(G)`

`SymmetricForms(G)`

`AntisymmetricForms(G)`

For an integral or rational matrix group G, return a basis for the space of G-linear forms or for the subspace of (anti-) symmetric forms respectively.

The first form returned by `InvariantForms` and `SymmetricForms` will be positive definite.

`InvariantForms(G, n)`

`SymmetricForms(G, n)`

`AntisymmetricForms(G, n)`

For an integral or rational matrix group G, return a sequence consisting of $n \geq 0$ G-invariant (symmetric or antisymmetric) bilinear forms for G.

`NumberOfInvariantForms(G)`

`NumberOfSymmetricForms(G)`

`NumberOfAntisymmetricForms(G)`

For an integral or rational matrix group G or a G-lattice L, return the dimension of the space of (symmetric or anti-symmetric) invariant bilinear forms for G.

The algorithm uses a modular method which is much faster than the actual computation of the forms.

62.3 Endomorphisms

`EndomorphismRing(G)`

For an integral or rational matrix group G, return the endomorphism ring (i.e. the commuting algebra) of G as a subalgebra of $M_n(\mathbf{Z})$ or $M_n(\mathbf{Q})$ respectively.

`CentreOfEndomorphismRing(G)`

`CentreOfEndomorphismAlgebra(G)`

For an integral or rational matrix group G, return the center of the endomorphism ring (i.e. the commuting algebra) of G as a subalgebra of $M_n(\mathbf{Z})$ or $M_n(\mathbf{Q})$ respectively.

`DimensionOfEndomorphismRing(G)`

Return the dimension of the endomorphism ring of an integral or rational matrix group G by a modular method.

`DimensionOfCentreOfEndomorphismRing(G)`

Return the dimension of the centre of the endomorphism ring of an integral or rational matrix group G by a modular method.

`Endomorphisms(G, n)`

For an integral or rational matrix group G, return a sequence containing n independent endomorphisms of G. n must be in the range $[0..d]$, where d is the dimension of the endomorphism ring of G.

`CentralEndomorphisms(G, n)`

For an integral or rational matrix group G, return a sequence containing n independent central endomorphisms of G. n must be in the range $[0..d]$, where d is the dimension of the centre of the endomorphism ring of G.

62.4 New Groups From Others

`BravaisGroup(G)`

For a finite integral matrix group G, compute its Bravais group which is the integral group fixing all symmetric bilinear forms fixed by G.

`IntegralGroup(G)`

Return the action of the finite rational matrix group G on an invariant lattice as an integral matrix group, thus giving an equivalent integral group H, together with the transformation matrix T from the standard lattice to the invariant lattice. Thus $H = T \cdot G \cdot T^{-1}$.

62.5 Perfect Forms and Normalizers

`PerfectForms(G)`

`Limit` RngIntElt *Default :* ∞

A positive definite symmetric G-invariant form F is called G-perfect if for every nonzero symmetric G-invariant form F' there exists some shortest vector x of F such that $F'x^{tr}x$ has nonzero trace.

The normalizer of the Bravais group of G in $\mathrm{GL}_n(\mathbf{Z})$ acts on the set of integral G-perfect forms whose entries have GCD 1 and the number of orbits is finite. This function returns a sequence of representatives of these orbits.

If `Limit` is set to a positive integer m, then the algorithm stops after m orbits have been enumerated.

`NormalizerGLZ(G)`

`CentralizerGLZ(G)`

`IsBravais` BoolElt *Default :* `false`

Given a finite subgroup G of $\mathrm{GL}_n(\mathbf{Z})$, returns the normalizer or centralizer of G in $\mathrm{GL}_n(\mathbf{Z})$.

If G is know to be equal to its Bravais group, one can set `IsBravais` to `true` to speed up the computation.

The algorithm employed is a variation of Opgenorth's normalizer algorithm [Opg01].

62.6 Conjugacy

`ZClasses(G)`

`Homogeneously` BoolElt *Default :* `false`

Given a finite integral or rational matrix group G, its $\mathrm{GL}_n(\mathbf{Q})$-conjugacy class splits into finitely many $\mathrm{GL}(n,\mathbf{Z})$-conjugacy classes. Representatives of these classes are constructed as the action of G on some G-invariant sublattices. More precisely, the $\mathrm{GL}(n,\mathbf{Z})$-conjugacy classes are in bijection with the orbits of G-invariant lattices under the normalizer N of G in $\mathrm{GL}(n,\mathbf{Q})$.

A G-lattice L' belongs to a G-lattice L if $L = \sum_i L'e_i$ where $e_1,\ldots,e_r$ denote the central idempotents of the endomorphism ring of G. Further, L is called *homogeneously decomposable* if L belongs to itself.

The algorithm will first compute representatives $L_1,\ldots,L_k$ of the orbits of homogeneously decomposable G-lattices under the action of N.

In a second step, it will then compute the G-lattices $L_{i,j}$ belonging to L_i up to the action of N.

The second return value will then consist of a sequence of k sequences $T_1,\ldots,T_k$. The first element $T_i[1]$ is the basis matrix of L_i, the following entries are basis matrices of the lattices $L_{i,j}$.

The first return value is a sequence of integral matrix groups describing the action of G on the lattices $L_{1,1}, L_{1,2},\ldots$. Hence these groups correspond to the $\mathrm{GL}_n(\mathbf{Z})$-conjugacy classes of G.

If `Homogeneously` is set to `true`, the function will only compute the homogeneously decomposable lattices $L_1,\ldots,L_k$ and the corresponding matrix groups. (If G is reducible, this option is much faster, but will not yield all conjugacy classes / orbits of lattices.)

`IsGLZConjugate(G, H)`

Tests whether the finite integral matrix groups G and H are conjugate in $\mathrm{GL}_n(\mathbf{Z})$. If so, a matrix x such that $G^x = H$ is also returned.

`IsBravaisEquivalent(G, H)`

Given two finite integral matrix groups G and H, tests whether their Bravais groups $B(G)$ and $B(H)$ are conjugate in $\mathrm{GL}_n(\mathbf{Z})$. If so, a matrix x such that $B(G)^x = B(H)$ is also returned.

Note that this function does not need to compute the Bravais groups and hence it is faster than calling `IsGLZConjugate` on the Bravais groups directly.

If G and H are known to be Bravais groups, this function is usually more efficient than calling `IsGLZConjugate`.

`IsGLQConjugate(G, H)`

`Al` MonStgElt *Default :*

Tests whether the finite rational matrix groups G and H are conjugate in $\mathrm{GL}_n(\mathbf{Q})$. If so, a matrix x such that $G^x = H$ is also returned.

There are currently two algorithms available. If the optional parameter `Al` equals `"Aut"`, Magma will use the `GModule`-machinery together with the outer automorphism group of H. If `Al` is set to `"ZClasses"`, Magma splits the $\mathrm{GL}(n, \mathbf{Q})$-conjugacy class of H into $\mathrm{GL}_n(\mathbf{Z})$-conjugacy classes and then decides whether an integral copy of G lies in one of these classes by several calls to `IsGLZConjugate`.

If `Al` is not provided, a sensible choice is made by the system.

62.7 Conjugacy Tests for Matrices

Given two $n \times n$ matrices A and B with rational or integral entries, Magma can test whether A is conjugate to B in $\mathrm{GL}_n(Z)$.

Currently, the implementation is limited to the cases where A, B have finite order or where $n = 2$. This limitation will be removed in future versions.

`IsGLZConjugate(A, B)`

`IsSLZConjugate(A, B)`

Tests whether two rational or integral matrices A and B are conjugate in $\mathrm{GL}_n(\mathbf{Z})$ or $\mathrm{SL}_n(\mathbf{Z})$. If so, a matrix x such that $A^x = B$ is also returned.

`CentralizerGLZ(A)`

Given a rational or integral matrix A, this function returns its centralizer in $\mathrm{GL}_n(\mathbf{Z})$. The current implementation is limited to the cases where either A has finite order or A is a 2×2 matrix.

62.8 Examples

Example H62E1

We split the $\mathrm{GL}_3(\mathbf{Q})$-conjugacy class of the following faithful representation of the dihedral group with 12 elements.

```
> G := MatrixGroup< 3, Integers() |
>  [ 1, -1, 0, 0, -1, 0, 0, 0, 1 ],
>  [ 1, -1, 0, 1, 0, 0, 0, 0, -1 ] >;
> Z, T:= ZClasses(G);
> #Z;
3
> < #t : t in T >;
```

```
<1, 2>
```

So there are 2 classes of homogeneously decomposable lattices represented by `T[1,1]` and `T[2,1]`. The third lattice `T[2,2]` belongs to `T[2,1]` as we check.

```
> Q := Rationals();
> GQ := ChangeRing(G, Q);
> Ids := CentralIdempotents(EndomorphismRing(GQ));
> L := VerticalJoin([ Matrix(Integers(), T[2,2] * i) : i in Ids]);
> Image(L) eq Image(Matrix(Integers(), T[2,1]));
true
```

Finally, we check that the 3 $\mathrm{GL}_3(\mathbf{Z})$-conjugacy classes stored in `Z` correspond to the 3 lattices in `T`.

```
> TT := &cat T;
> [ GQ eq ChangeRing(Z[i], Q)^(GL(3, Q) ! TT[i]) : i in [1..#Z] ];
[ true, true, true ]
```

Example H62E2

We test that the automorphism groups of the lattices B_8 and D_8 are conjugate in $\mathrm{GL}_8(\mathbf{Q})$ but not in $\mathrm{GL}_8(\mathbf{Z})$.

```
> G := AutomorphismGroup( Lattice("B", 8) );
> H := AutomorphismGroup( Lattice("D", 8) );
> ok, x := IsGLQConjugate(G, H); ok, x;
true
[ 1 -1  0  0  0  0  0  0]
[ 1 -1 -2  0  0  0  0  0]
[-1  1  2  2  2  2  2  2]
[ 1  1  0  0  0  0  0  0]
[-1  1  2  2  2  2  2  0]
[ 1 -1 -2 -2 -2  0  0  0]
[-1  1  2  2  2  2  0  0]
[-1  1  2  2  0  0  0  0]
> Determinant(x);
-128
> IsGLZConjugate(G,H);
false
```

Example H62E3

Let C be the companion matrix of the fifth cyclotomic polyomial. We find a unimodular matrix that induces the automorphism $C -> C^2$.

```
> C:= CompanionMatrix(CyclotomicPolynomial(5));
> ok, h:= IsGLZConjugate(C, C^2); ok;
true
> C^2 eq h^-1 * C * h;
```

```
true
```

We now check by hand that this automorphism cannot be realized by a matrix of determiant 1.

```
> Determinant(h);
-1
> G:= CentralizerGLZ(C);
> [ Determinant(g) : g in Generators(G) ];
[1, 1, 1]
```

Of course, we could also just ask:

```
> IsSLZConjugate(C, C^2);
false
```

62.9 Bibliography

[**Opg01**] J. Opgenorth. Dual Cones and the Voronoi Algorithm. *Exp. Math.*, 10(4):599–608, 2001.

[**OPS98**] J. Opgenorth, W. Plesken, and T. Schulz. Crystallographic Algorithms and Tables. *Acta Crystallographica*, A54:517–531, 1998.

63 FINITE SOLUBLE GROUPS

Chapter 63
FINITE SOLUBLE GROUPS

63.1 Introduction

Any finite soluble group has a subnormal series with cyclic factors. Such a series gives rise to various polycyclic presentations. These polycyclic presentations are useful because the word problem in such presentations can be solved in an algorithmic fashion. In MAGMA, we use the specific form called a *power-conjugate presentation (pc-presentation)*, which is described below. The MAGMA category of groups represented by a power-conjugate presentation (pc-groups for short) is called `GrpPC`.

This chapter describes how to use polycyclic presentations to compute with p-groups and other finite soluble groups in MAGMA. While most functions apply to any soluble group, a small number of functions specific to p-groups are identified in the text.

Over the past two decades a considerable body of efficient algorithms has been developed for computing with soluble groups defined in terms of pc-presentations. It is recommended that the `GrpPC` representation of a soluble group be used whenever intensive calculation with that group is necessary.

63.1.1 Power-Conjugate Presentations

Let G be a finite soluble group. A presentation for G of the form

$$< a_1, \ldots, a_n \mid a_j^{p_j} = w_{jj}, \quad 1 \leq j \leq n, \quad a_j^{a_i} = w_{ij}, \quad 1 \leq i < j \leq n >$$

where

(i) p_j is the least prime such that $a_j^{p_j} \in < a_{j+1}, \ldots, a_n >$ for $j < n$, and $a_j^{p_j}$ is the identity for $j = n$, and

(ii) w_{ij} is a word in the generators $a_{i+1}, \ldots, a_n$, will be called a *power-conjugate presentation (pc-presentation)* for G. The generators of G corresponding to $a_1, \ldots, a_n$ in this presentation are known as a *power-conjugate generating sequence (pc-generators)* for G.

It is easy to show that every finite soluble group possesses a pc-presentation. If such a presentation satisfies a certain additional condition (the *consistency condition*) then every element a of G can be written uniquely in the *normal form*

$$a_1^{\alpha_1} \ldots a_n^{\alpha_n}, 0 \leq \alpha_i < p_i \quad \text{for } i = 1, \ldots, n.$$

Given such a pc-presentation for G there exists an algorithm (the *collection algorithm*), which given an arbitrary word in the pc-generators $a_1, \ldots, a_n$, will determine the corresponding normal word. In particular, collection can be used to compute the normal word which is equal to the product of two given normal words, thus implementing the group multiplication.

63.2 Creation of a Group

A user can create a GrpPC representation of a finite soluble group in a variety of ways. There are several built-in construction functions for creating standard examples such as cyclic or dihedral groups. For greater flexibility, it is possible to define a group directly from a power-commutator presentation. One can also build new groups out of old groups using standard constructions such as direct product. Finally, there are several conversion functions which will automatically compute a pc-presentation for an existing soluble group in some other category (such as permutation group or matrix group). We will start with the first two styles of construction and describe the remaining two in later sections.

In each case, regardless of how the group was originally defined, MAGMA will store the group internally as a pc-presentation and will display the pc-presentation whenever the group is printed. Normally when printing a pc-presentation, trivial conjugate relations are omitted. In the case of a p-group, then trivial power relations (those indicating that a generator has order p) are also omitted. The one exception to this policy is in the case of elementary abelian p-groups (which would have no relations displayed under the above policies). In the elementary abelian case, MAGMA will display the power relations, even though they are trivial.

63.2.1 Construction Functions

The simplest method of producing a pc-presentation for a group is to use one of the built-in construction functions. By specifying the category GrpPC as the first parameter of each function, we produce the desired representation.

It is also possible to obtain a pc-presentation for many small soluble groups by using the function SmallGroup described in Chapter 66.

`CyclicGroup(GrpPC, n)`

The cyclic group of order n as a pc-group.

`AbelianGroup(GrpPC, Q)`

Construct the abelian group defined by the sequence $Q = [n_1, \ldots, n_r]$ of positive integers as a pc-group. The function returns the abelian group which is the direct product of the cyclic groups $C_{n_1} \times C_{n_2} \times \cdots \times C_{n_r}$.

`DihedralGroup(GrpPC, n)`

The dihedral group of order $2 * n$ as a pc-group.

`ExtraSpecialGroup(GrpPC, p, n : parameters)`

Given a small prime p and a small positive integer n, construct an extra-special group G of order p^{2n+1} in the category GrpPC. The isomorphism type of G may be selected using the parameter Type.

Type MONSTGELT *Default :* "+"

Possible values for this parameter are "+" (default) and "−".

If `Type` is set to "+", the function returns, for $p = 2$, the central product of n copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p.

If `Type` is set to "−", the function returns for $p = 2$ the central product of a quaternion group of order 8 and $n - 1$ copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p^2.

Example H63E1

A pc-representation for the cyclic group C_{12} can be computed as follows.

```
> G := CyclicGroup(GrpPC, 12);
```

We can then check various properties of G.

```
> Order(G);
12
> IsAbelian(G);
true
> IsSimple(G);
false
```

If we simply print G, we will see the presentation which MAGMA has generated for this group.

```
> G;
GrpPC : G of order 12 = 2^2 * 3
PC-Relations:
    G.1^2 = G.2,
    G.2^2 = G.3,
    G.3^3 = Id(G)
```

Or, we could build a slightly different group.

```
> H := AbelianGroup(GrpPC, [2,2,3]);
> Order(H);
12
> IsCyclic(H);
false
```

63.2.2 Definition by Presentation

While the standard construction functions are convenient, most groups cannot be defined in that way. Complete flexibility in defining a soluble group can be obtained by directly specifying the group's pc-presentation.

One uses a power-conjugate presentation to define a soluble group by means of the `PolycyclicGroup` constructor, or the `quo` constructor for finitely presented groups.

`PolycyclicGroup< x1, ..., xn | R : parameters >`

`Check`	BOOLELT	*Default :* `true`
`ExponentLimit`	RNGINTELT	*Default :* 20
`Class`	MONSTGELT	*Default :*

Construct the soluble group G defined by the power-conjugate presentation $< x_1, \ldots, x_n \mid R >$.

The construct $x_1, \ldots, x_n$ defines names for the generators of G that are local to the constructor, i.e. they are used when writing down the relations to the right of the bar. However, no assignment of *values* to these variables is made. If the user wants to refer to the generators by these (or other) names, then the *generators assignment* construct must be used on the left hand side of an assignment statement.

The construct R denotes a list of pc-relations. Thus, an element of R must be one of:

(a) A *power relation* $a_j^{p_j} = w_{jj}$, $1 \leq j \leq n$, where w_{jj} is 1 or a word in generators $a_{j+1}, \ldots, a_n$ for $j < n$, and $w_{jj} = 1$ for $j = n$, and p_j a prime.

(b) A *conjugate relation* $a_j^{a_i} = w_{ij}$, $1 \leq i < j \leq n$, where w_{ij} is a word in the generators $a_{i+1}, \ldots, a_n$.

(c) A *power* $a_j^{p_j}$, $1 \leq j \leq n$ and p_j a prime, which is treated as the *power relation* $a_j^{p_j} = Id(F)$.

(d) A set of (a) – (c).

(e) A sequence of (a) – (c).

Note the following points:

(i) A power relation must be present for each generator a_i, $i = 1, \ldots, n$;

(ii) Conjugate relations involving commuting generators (i.e. of the form $y^x = y$) may be omitted;

(iii) The words w_{ij} must be in normal form.

In addition, one can alternatively specify a power-commutator presentation using commutator relations rather than conjugate relations.

(b') A *commutator relation* $(a_j, a_i) = w_{ij}$, $1 \leq i < j \leq n$, where w_{ij} is a word in the generators $a_{i+1}, \ldots, a_n$. However, commutators and conjugates cannot be mixed in a single presentation.

A map f from the free group of rank n to G is returned as well.

The parameters `Check` and `ExponentLimit` may be used. `Check` indicates whether or not the presentation is checked for consistency. `ExponentLimit` determines the amount of space that will be used by the group to speed calculations. Given `ExponentLimit := e`, the group will precompute and store normal words for appropriate products $a^i * b^j$ where a and b are generators and i and j are in the range 1 to e.

If the construction of an object in the category `GrpPC` fails because R is not a valid power-conjugate presentation, an attempt is made to construct a group in the

category GrpGPC (cf. Chapter 72). This feature can be turned off by setting the parameter Class to "GrpPC"; an invalid power-conjugate presentation then causes a runtime error. Since, by default, the constructor always returns a group in the category GrpPC if possible, this is the only effect of setting the parameter Class to "GrpPC".

quo< GrpPC : F | R : *parameters* >

Check	BoolElt	*Default :* true
ExponentLimit	RngIntElt	*Default :* 20

Given a free group F of rank n with generating set X, and a collection R of pc-relations on X, construct the soluble group G defined by the power-conjugate presentation $< X|R >$.

The construct R denotes a list of pc-relations. The syntax and semantics for the relations clause is identical to that appearing in the PolycyclicGroup-construct.

This constructor returns a pc-group because the category GrpPC is stated. If no category were stated, it would return an fp-group.

The parameters Check and ExponentLimit may be used as described in the PolycyclicGroup-construct.

The natural homomorphism, $F \to G$, is also returned.

Example H63E2

Consider the group of order 80 defined by the presentation

$$< a, b, c, d, e \mid a^2 = c, b^2, c^2 = e, d^5, e^2, b^a = b * e, c^a = c, c^b = c,$$
$$d^a = d^2, d^b = d, d^c = d^4, e^a = e, e^b = e, e^c = e, e^d = e > .$$

Giving the relations in the form of a list, this presentation would be specified as follows:

```
> G<a,b,c,d,e> := PolycyclicGroup<a, b, c, d, e |
>  a^2 = c, b^2, c^2 = e, d^5, e^2,
>  b^a = b*e, d^a = d^2, d^c = d^4 >;
```

Starting from a free group and giving the relations in the form of a set of relations, this presentation would be specified as follows:

```
> F<a,b,c,d,e> := FreeGroup(5);
> rels := { a^2 = c, b^2 = Id(F), c^2 = e, d^5 = Id(F), e^2 = Id(F),
>  b^a = b*e, d^a = d^2, d^c = d^4 };
> G<a,b,c,d,e> := quo< GrpPC : F | rels >;
```

Notice that here we have redefined the variables $a, \ldots, e$ to be the pc-generators in G. Thus, when G is printed, Magma displays the following presentation:

```
> G;
GrpPC : G of order 80 = 2^4 * 5
PC-Relations:
    a^2 = c,
```

```
    b^2 = Id(G),
    c^2 = e,
    d^5 = Id(G),
    e^2 = Id(G),
    b^a = b * e,
    d^a = d^2,
    d^c = d^4
> Order(G);
80
> IsAbelian(G);
false
```

63.2.3 Possibly Inconsistent Presentations

The `PolycyclicGroup` and `quo` constructors accept a parameter `Check` which enables the user to suppress the automatic consistency checking for input presentations. This is primarily intended to be used when it is certain that the input presentation is consistent, in order to save time. For instance, the presentation may have been generated from some other reliable program, or even from an earlier MAGMA session. This parameter should be used with care, since all of the MAGMA functions assume that every `GrpPC` group is consistent. The user will encounter numerous bizarre results if an attempt is made to compute with an inconsistent presentation.

On occasion, a user may wish to "try out" a series of pc-presentations, some of which may not be consistent. The `Check` parameter can be used, along with the function `IsConsistent`, to test a presentation for consistency.

`IsConsistent(G)`

Returns `true` if G has a consistent presentation, `false` otherwise.

Example H63E3

The following example demonstrates generating a family of presentations, and then checking consistency. Of course, it is easy to predict the outcome in this simple example.

```
> F := FreeGroup(2);
> for p in [n: n in [3..10] | IsPrime(n)] do
>   r := [F.1^3=Id(F), F.2^p=Id(F), F.2^F.1=F.2^2];
>   G := quo<GrpPC: F | r: Check:=false>;
>   if IsConsistent(G) then
>     print "For p=",p," the group is consistent.";
>   else
>     print "For p=",p," the group is inconsistent.";
>   end if;
> end for;
For p= 3  the group is inconsistent.
For p= 5  the group is inconsistent.
```

```
For p= 7  the group is consistent.
```

63.3 Basic Group Properties

63.3.1 Infrastructure

The functions described here provide access to basic information stored for a pc-group G.

`G . i`

The i-th pc-generator for G. A negative subscript indicates that the inverse of the generator is to be created. $G.0$ is `Identity(G)`.

`Generators(G)`

A set containing the defining generators for G. If G is a p-group, this is guaranteed to be a minimal set of generators. For non-p-groups, this will be the set of pc-generators.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for G.

`PCGenerators(G)`

An indexed set containing the pc-generators for G.

`NumberOfPCGenerators(G)`

`NPCGenerators(G)`

`NPCgens(G)`

The number of pc-generators for G.

`PCPrimes(G)`

A sequence $[p_1, \ldots, p_n]$ containing the primes associated with the pc-generators of G. The i-th term of the sequence contains the prime associated with generator a_i of G for $i = 1, \ldots, n$.

63.3.2 Numerical Invariants

MAGMA has built-in functions to compute the order and exponent of a group.

`Order(G)`
`#G`

The order of the group G, returned as an ordinary integer.

`FactoredOrder(G)`

The factored order of the group G.

`Exponent(G)`

The exponent of the group G.

63.3.3 Predicates

MAGMA has built-in functions to check standard group properties.

`IsAbelian(G)`

Returns `true` if the group G is abelian, `false` otherwise.

`IsCyclic(G)`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group G is elementary abelian, `false` otherwise.

`IsNilpotent(G)`

Returns `true` if the group G is nilpotent, `false` otherwise.

`IsPerfect(G)`

Returns `true` if the group G is perfect, `false` otherwise. A soluble group G is perfect only if it is trivial.

`IsSimple(G)`

Returns `true` if the group G is simple, `false` otherwise.

`IsSoluble(G)`
`IsSolvable(G)`

Returns `true` if the group G is soluble, `false` otherwise. It always returns the value `true` for a pc-group.

`IsTrivial(G)`

Returns `true` if the group G has order 1, `false` otherwise.

`IsSpecial(G)`

Given a p-group G, return `true` if G is special, `false` otherwise.

`IsExtraSpecial(G)`

Given a p-group G, return `true` if G is extra-special, `false` otherwise e.

Example H63E4

We use a presentation to define an extraspecial 3-group of exponent 9.

```
> E := PolycyclicGroup<a1,a2,b1,b2,z|a1^3,a2^3,b1^3=z,b2^3=z,
>   z^3,b1^a1=b1*z,b2^a2=b2*z>;
```

The sequence of base, exponent pairs from `FactoredOrder` shows us that the group has order 3^5.

```
> FactoredOrder(E);
[ <3, 5> ]
> Exponent(E);
9
```

As well as with the `Order` function, one can get the size of a group by using the `#` shorthand.

```
> D3 := DihedralGroup(GrpPC, 3);
> #D3;
6
> IsNilpotent(D3);
false
```

63.4 Homomorphisms

Arbitrary homomorphisms can be defined between pc-groups by using the `hom<>` constructor. For pc-groups, this constructor has features not generally available for user-defined homomorphisms. In addition to defining the map by giving images for the pc-generators, a homomorphism can be defined by giving images for any generating set of the domain. MAGMA will verify that the specified images define a homomorphism and will compute the kernel and inverse images for the defined map. Note that the value returned for an inverse image of an element is simply one element from the preimage, not the complete coset.

`hom< G -> H | L >`

`Check` BOOLELT *Default :* `true`

Construct a homomorphism $\phi : G \to H$ defined by the images specified by the list L.

The list L must be one of the following:

(a) a list, set, or sequence of 2-tuples $< g_i, h_i >$ (order not important);

(b) a list, set, or sequence of arrow pairs $g_i -> h_i$ (order not important);

(c) a list or sequence of images $h_1, \ldots, h_n$ (order is important).

The elements g_i and h_i must be elements of G and H, respectively, in each case. For the cases (a) and (b), the elements g_i must generate G and the homomorphism will satisfy $\phi(g_i) = h_i$. For case (c), n must be the number of pc-generators of G and the g_i are implicitly defined to be the pc-generators.

The parameter `Check` can be set to false in order to turn off the check that the map defined is a homomorphism. This should only be done when one is certain that the map is a homomorphism, since later results will most likely be incorrect if it is not.

`IsHomomorphism(G, H, L)`

This is a conditional form of the `hom`-constructor. The argument L must be a set or sequence of pairs (as in case (a) of the `hom`-constructor), or a sequence of images in H for the pc-generators of G (as in case (c) of the `hom`-constructor). If the specified images define a homomorphism, the value true and the resulting map are returned. Otherwise, false is returned.

`IdentityHomomorphism(G)`

The identity map from G to G.

`Kernel(f)`

Given a homomorphism f from one pc-group to another, return the kernel of f. This will be a pc-group which is a subgroup of the domain of f.

`Homomorphisms(G, H)`

Given finite *abelian* groups G and H, return a sequence containing all elements of $\mathrm{Hom}(G, H)$. The elements are returned as actual (MAGMA map type) homomorphisms. Note that this function simply uses `Hom`, transferring each element of the returned group to the actual MAGMA map type homomorphism.

Example H63E5

Let G be a pc-representation of the symmetric group S_4, and N be $O_2(G)$.

```
> G := PCGroup(Sym(4));
> G;
GrpPC : G of order 24 = 2^3 * 3
PC-Relations:
    G.1^2 = Id(G),
    G.2^3 = Id(G),
    G.3^2 = Id(G),
    G.4^2 = Id(G),
    G.2^G.1 = G.2^2,
    G.3^G.1 = G.3 * G.4,
    G.3^G.2 = G.4,
    G.4^G.2 = G.3 * G.4
> N := pCore(G,2);
> Order(N);
```

```
4
```

Let us define H to be a complement of N in G.

```
> H := sub<G|G.1*G.4,G.2*G.4>;
> Order(H);
6
> H meet N;
GrpPC of order 1
PC-Relations:
```

We now wish to define the projection homomorphism from G to H. This will map each element of N to the identity and each element of H to itself. We can define the map directly using these properties.

```
> pairs := [];
> for n in Generators(N) do
>   pairs cat:= [<G!n,Id(H)>];
> end for;
> for h in Generators(H) do
>   pairs cat:= [<G!h, h>];
> end for;
> proj := hom<G -> H|pairs>;
> proj;
Mapping from: GrpPC: G to GrpPC: H
> proj(G.1);
H.1
> proj(N);
GrpPC of order 1
PC-Relations:
```

We can also compute inverse images and can verify that N is the kernel of the map. Note that the preimage of a single element is just one element from the preimage, not the complete coset. Of course, one can use the kernel to compute the full coset if desired.

```
> y := (H.1)@@proj;
> y;
G.1
> Kernel(proj) eq N;
true
> {y*k: k in Kernel(proj)};
{ G.1 * G.3 * G.4, G.1 * G.4, G.1, G.1 * G.3 }
```

63.5 New Groups from Existing

`DirectProduct(G, H)`

The direct product K of the pc-groups G and H. The second argument returned is a sequence containing the inclusion maps $I_G : G \to K$ and $I_H : H \to K$. The third argument returned is a sequence containing the projection maps $P_G : K \to G$ and $P_H : K \to H$. Furthermore, the (user-) presentation of K is arranged so that the first pc-generators correspond to those of G and the remaining generators correspond to those of H.

`DirectProduct(Q)`

The direct product of pc-groups in the non-empty sequence Q, and the inclusion and projection maps.

`Extension(G, H, f)`

The split extension K of the pc-group G by the pc-group H, where the action of H on G is given by the homomorphism $\phi : H \to \mathrm{Aut}(G)$ specified by f. The extension K will have a normal subgroup $G^\sim$ isomorphic to G, while the quotient group $K/G^\sim$ is isomorphic to H.

The homomorphism ϕ is given by the sequence of maps f. Suppose that the pc-generators for H are $h_1, \ldots, h_s$. The i-th entry of f defines the action of h_i on G. That is, $f[i](x) = h_i^{-1} \cdot x \cdot h_i$, for $x \in G$.

`Extension(M, H)`

The split extension K of the G-module M by the pc-group H. We use the action of H on M to define the action of H on an elementary abelian p-group of order p^d where M is a d-dimensional module over $GF(p)$, p prime.

`Extension(G, H, f, t)`

The non-split extension K of the pc-group G by the pc-group H, where the action of H on G is given by the homomorphism $\phi : H \to \mathrm{Aut}(G)$ and the tails for H are given as the set of tuples t. The extension K will have a normal subgroup $G^\sim$ isomorphic to G, while the quotient group $K/G^\sim$ is isomorphic to H.

The homomorphism ϕ is given by the sequence of maps f. Suppose that the pc-generators for H are $h_1, \ldots, h_s$. The i-th entry of f defines the action of h_i on G. That is, $f[i](x) = h_i^{-1} \cdot x \cdot h_i$, for $x \in G$.

The specification of t involves giving the relations $h_j^{-1} h_i h_j = w_{ij}$, where w_{ij} is a word in K for $1 \le j < i \le s$. For $i = j$, we need the relation ${h_i}^{p_i} = w_{ii}$, where w_{ii} is a word in K for $1 \le i \le s$. Each w_{ij} is the RHS of the relation from H with the tail x_{ij}. The tails are given by the sequence t in the order $t = [x_{11}, x_{21}, x_{22}, x_{31}, \ldots, x_{ss}]$. Alternatively, t can be given as a set of tuples $< i, j, x_{ij} >$ for non-trivial x_{ij}.

Note that if $x_{ij} =$`Id(G)`, for $1 \le i \le s$ and $1 \le j \le i$, then K will just be the split extension of G and H.

`Extension(M, H, t)`

The non-split extension K of the G-module M by the pc-group H. We use the action of H on M to define the action of H on an elementary abelian p-group of order p^d where M is a d-dimensional module over $GF(p)$, p prime.

The specification of t is similar to that for t in the preceding description.

`IsExtension(G, H, f)`

`IsExtension(M, H)`

`IsExtension(G, H, f, t)`

`IsExtension(M, H, t)`

For each `Extension` variation, there is a corresponding function `IsExtension` which attempts to construct the specified group and returns a boolean value indicating whether or not the construction succeeded. If the construction succeeds, the extension group is also returned.

The `Extension` functions will generate a runtime error if the specified construction is not legal. The `IsExtension` function allows the user to detect this error condition and continue.

`WreathProduct(G, H)`

The wreath product of the pc-groups G and H, where the regular permutation representation of H is used to define the action.

`WreathProduct(G, H, f)`

The wreath product of the pc-groups G and H where the action of H is given by f, which may be either a homomorphism from H into a permutation group P or a sequence of permutations defining a homomorphism from H into P. If f is a sequence, the homomorphism $\phi : H \to P$ is defined by $H.i \to f[i]$ for $i = 1, \ldots, s$.

Example H63E6

To demonstrate some of the versions of `Extension` we first build a split extension of a cyclic group of order 4 acting on an elementary abelian group of order 9.

```
> C4 := CyclicGroup(GrpPC,4);
> E9 := AbelianGroup(GrpPC,[3,3]);
> f1 := hom<E9->E9|[E9.1*E9.2^2, E9.1^2*E9.2^2]>;
> f2 := hom<E9->E9|[E9.1^2,E9.2^2]>;
> G := Extension(E9,C4,[f1,f2]);
> G;
GrpPC : G of order 36 = 2^2 * 3^2
PC-Relations:
    G.1^2 = G.2,
    G.2^2 = Id(G),
    G.3^3 = Id(G),
    G.4^3 = Id(G),
```

```
    G.3^G.1 = G.3 * G.4^2,
    G.3^G.2 = G.3^2,
    G.4^G.1 = G.3^2 * G.4^2,
    G.4^G.2 = G.4^2
```

Then, we define a module for this group and use it to build a nonsplit extension.

```
> MR := MatrixRing(GF(3),2);
> m1 := MR![1,1,1,2];
> m2 := MR![2,0,0,2];
> V := GModule(G,[m1,m2,Id(MR),Id(MR)]);
> IsIrreducible(V);
true
> v0 := V!0;
> tails := [v0,v0,v0,v0,V![1,0],V![2,0],V![1,2],V![0,2],v0,V![0,1]];
> H := Extension(V,G,tails);
> H;
GrpPC : H of order 324 = 2^2 * 3^4
PC-Relations:
    H.1^2 = H.2,
    H.2^2 = Id(H),
    H.3^3 = H.5^2,
    H.4^3 = H.6,
    H.5^3 = Id(H),
    H.6^3 = Id(H),
    H.3^H.1 = H.3 * H.4^2,
    H.3^H.2 = H.3^2 * H.5,
    H.4^H.1 = H.3^2 * H.4^2 * H.5 * H.6^2,
    H.4^H.2 = H.4^2 * H.6^2,
    H.5^H.1 = H.5 * H.6,
    H.5^H.2 = H.5^2,
    H.6^H.1 = H.5 * H.6^2,
    H.6^H.2 = H.6^2
```

Notice that the relations of H involving the first four generators are those of G with the specified tails appended. We are then ready to compute various properties of H.

```
> [N`order:N in NormalSubgroups(H)];
[ 1, 9, 81, 162, 324 ]
```

Example H63E7

In this example we verify an example of Cossey and Hawkes in [CH00]. The paper shows that the largest size of a conjugacy class in an abelian by nilpotent finite group is at least as large as the product of the largest class sizes for the Sylow subgroups. The example is a group having derived length 3 in which this fails.
We start with a dihedral group of order 10 acting on a cyclic group of order 8.

```
> E := DihedralGroup(GrpPC,5);
```

```
> A := CyclicGroup(GrpPC,8);
```

Define an action of E on A and create the split extension.

```
> f1 := hom<A->A|A.1->(A.1)^-1>;
> f2 := hom<A->A|A.1->A.1>;
> H := Extension(A, E, [f1, f2]);
```

Then construct a certain H-module...

```
> QH := SylowSubgroup(H,2);
> t := TrivialModule(QH, FiniteField(5));
> B := Induction(t, H);
```

...and form the split extension of H acting on that module.

```
> G := Extension(B, H);
> print G;
GrpPC : G of order 250000 = 2^4 * 5^6
PC-Relations:
    G.1^2 = Id(G),
    G.2^5 = Id(G),
    G.3^2 = G.4,
    G.4^2 = G.5,
    G.5^2 = Id(G),
    G.6^5 = Id(G),
    G.7^5 = Id(G),
    G.8^5 = Id(G),
    G.9^5 = Id(G),
    G.10^5 = Id(G),
    G.2^G.1 = G.2^4,
    G.3^G.1 = G.3 * G.4 * G.5,
    G.4^G.1 = G.4 * G.5,
    G.6^G.2 = G.10,
    G.7^G.1 = G.10,
    G.7^G.2 = G.6,
    G.8^G.1 = G.9,
    G.8^G.2 = G.7,
    G.9^G.1 = G.8,
    G.9^G.2 = G.8,
    G.10^G.1 = G.7,
    G.10^G.2 = G.9
> print DerivedLength(G);
3
```

Now check the relevant class sizes.

```
> P := SylowSubgroup(G,5);
> Q := SylowSubgroup(G,2);
> print  Maximum({x[2]:x in Classes(G)});
1250
```

```
> print  Maximum({x[2]:x in Classes(P)});
625
> print  Maximum({x[2]:x in Classes(Q)});
4
```

Note that 1250 is less than the product 625*4.

63.6 Elements

Elements of a pc-group are written in terms of the generators. The pc-generators of a group G can always be written as `G.1`, `G.2`, ... Any variables naming the generators, either assigned during the definition of the group, or later using standard assignment statements, can also be used to express the generators. An arbitrary element can be written as a word in the generators using the various element operations.

63.6.1 Definition of Elements

A *word* is defined inductively as follows:

(i) A generator is a word;

(ii) The expression (u) is a word, where u is a word;

(iii) The product $u * v$ of the words u and v is a word;

(iv) The conjugate u^v of the word u by the word v is a word (u^v expands into the word $v^{-1} * u * v$);

(v) The power of a word u^n, where u is a word and n is an integer, is a word;

(vi) The commutator (u, v) of the words u and v is a word ((u, v) expands into the word $u^{-1} * v^{-1} * u * v$).

A group element is always printed by MAGMA as a normal word in the pc-generators of its parent group.

It is also possible to create an element of a group G from its exponent vector. That is, the sequence $[e_1, e_2, \ldots, e_n]$ corresponds to the element $G.1^{e_1} * G.2^{e_2} * \cdots * G.n^{e_n}$. The coercion operator `!` is used to convert the sequence to the element.

`G ! Q`

Given the pc-group G and a sequence Q of length n, containing the distinct positive integers α_i, $0 \leq \alpha_i < p_i$ for $i = 1, \ldots, n$, construct the element x of G given by

$$x = a_1^{\alpha_1} \ldots a_n^{\alpha_n}, \quad 0 \leq \alpha_i < p_i \quad \text{for } i = 1, \ldots, n.$$

`ElementToSequence(x)`

`Eltseq(x)`

Given an element x belonging to the pc-group G, where

$$x = a_1^{\alpha_1} \ldots a_n^{\alpha_n}, \quad 0 \leq \alpha_i < p_i \quad \text{for } i = 1, \ldots, n,$$

return the sequence Q of n integers defined by $Q[i] = \alpha_i$, for $i = 1, \ldots, n$.

Identity(G)

Id(G)

G ! 1

Construct the identity element of the pc-group G.

Example H63E8

Given a pc-group, we can define elements as words in the generators or as more general expressions.

```
> G := PolycyclicGroup<a,b,c|a^3,b^2,c^2,b^a=c,c^a=b*c>;
> G;
GrpPC : G of order 12 = 2^2 * 3
PC-Relations:
    G.1^3 = Id(G),
    G.2^2 = Id(G),
    G.3^2 = Id(G),
    G.2^G.1 = G.3,
    G.3^G.1 = G.2 * G.3
> x := G.1^2*G.3;
> x;
G.1^2 * G.3
> x^2;
G.1 * G.2 * G.3
> x^3;
Id(G)
```

MAGMA will print the element in normal form even if it is not entered that way.

```
> G.2*G.1;
G.1 * G.3
```

When coercing a sequence into a group element, the sequence is always interpreted as an exponent vector for a normal word.

```
> y := G![0,1,1];
> y;
G.2 * G.3
> x*y;
G.1^2 * G.2
> y*x;
G.1^2
> (x,y);
G.2
```

An element can also be converted into a sequence.

```
> x^y;
G.1^2 * G.2 * G.3
> Eltseq(x^y);
[ 2, 1, 1 ]
```

63.6.2 Arithmetic Operations on Elements

New elements can be computed from existing elements using standard operations.

`g * h`

Product of the element g and the element h, where g and h belong to some common subgroup G of a pc-group U. If g and h are given as elements belonging to the same proper subgroup G of U, then the result will be returned as an element of G; if g and h are given as elements belonging to distinct subgroups H and K of U, then the product is returned as an element of G, where G is the smallest subgroup of U known to contain both elements.

`g *:= h`

Replace g with the product of element g and element h.

`g ^ n`

The n-th power of the element g, where n is a positive or negative integer.

`g ^:= n`

Replace g with the n-th power of the element g.

`g / h`

Quotient of the element g by the element h, i.e. the element $g * h^{-1}$. Here g and h must belong to some common subgroup G of a pc-group U. The rules for determining the parent group of g/h are the same as for $g * h$.

`g /:= h`

Replace g with the quotient of the element g by the element h.

`g ^ h`

Conjugate of the element g by the element h, i.e. the element $h^{-1} * g * h$. Here g and h must belong to some common subgroup G of a pc-group U. The rules for determining the parent group of g^h are the same as for $g * h$.

`g ^:= h`

Replace g with the conjugate of the element g by the element h.

`(g1, ..., gn)`

Given the n words $g_1, \ldots, g_n$ belonging to some common subgroup G of a pc-group U, return the commutator. If $g_1, \ldots, g_n$ are given as elements belonging to the same proper subgroup G of U, then the result will be returned as an element of G; if $g_1, \ldots, g_n$ are given as elements belonging to distinct subgroups of U, then the product is returned as an element of G, where G is the smallest subgroup of U known to contain all elements. Commutators are *left-normed*, so that they are evaluated from left to right.

63.6.3 Properties of Elements

`Order(x)`

Order of the element x.

`Parent(x)`

The parent group G of the element x.

63.6.4 Predicates for Elements

Elements in the same group can be compared using `eq` and `ne`.

`g eq h`

Given elements g and h belonging to a common pc-group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given elements g and h belonging to a common pc-group, return `true` if g and h are distinct elements, `false` otherwise.

`IsIdentity(g)`
`IsId(g)`

Returns `true` if g is the identity element, `false` otherwise.

`IsConjugate(G, g, h)`

Given a group G and elements g and h belonging to G, return the value `true` if g and h are conjugate in G. The function also returns a second value in the event that the elements are conjugate: an element z such that $g^z = h$.

Example H63E9

We check if one element commutes with another.

```
> G<a,b,c> := PolycyclicGroup<a,b,c|a^3,b^2,c^2,b^a=c,c^a=b*c>;
> b^a eq b;
false
```

The same information can also be obtained by checking the commutator.

```
> IsIdentity((b,a));
false
```

If we assign the result of `IsConjugate` to a single variable, it will store the boolean result.

```
> r := IsConjugate(G, c, b);
> r;
```

```
true
```

If we simply print `IsConjugate`, the boolean value and the conjugating element (if any) are displayed. On the other hand, using the multiple assignment, we can capture both of those values.

```
> IsConjugate(G, c, b);
true a^2
> r, x := IsConjugate(G, c, b);
> x, r;
a^2 true
> c^x;
b
```

63.6.5 Set Operations

These functions allow one to work with the set of elements of G, possibly without much knowledge of the structure of G.

`NumberingMap(G)`

A bijective mapping from the group G onto the set of integers $\{1...|G|\}$. The actual mapping depends upon the current presentation for G.

`Random(G)`

An element, randomly chosen, from the group G. This function uses an entirely different procedure than that used by `RandomProcess` (see below). A group element is chosen with uniform probability by generating (pseudo-)random integers in the proper range to form a legal exponent vector for G. The corresponding element is returned. This is an extremely efficient process and is the recommended method for producing random elements of a pc-group.

`RandomProcess(G)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 20

Create a process to generate randomly chosen elements from the group G. The process uses an 'expansion' procedure to construct a set of elements corresponding to fairly long words in the generators of G. At all times, N elements are stored where N is the maximum of the specified value for `Slots` and `Ngens`$(G) + 1$. Initially, these are simply the generators of G and products of pairs of generators of G. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call chooses an element x stored by the process and returns it, replacing x with the product of x and another random element (multiplied on the left or the right). Setting `Scramble := m` causes m such operations to be performed initially.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the finite group G, construct a random element of G by forming a random product over the expanded generating set constructed when the process was created.

`Representative(G)`

`Rep(G)`

A representative element of G. For a pc-group, this always returns the identity element.

Example H63E10

The `NumberingMap` function assigns a number to each group element.

```
> G := DihedralGroup(GrpPC,4);
> num_map := NumberingMap(G);
> for x in G do
>   print x,"->",num_map(x);
> end for;
Id(G) -> 1
G.3 -> 2
G.2 -> 3
G.2 * G.3 -> 4
G.1 -> 5
G.1 * G.3 -> 6
G.1 * G.2 -> 7
G.1 * G.2 * G.3 -> 8
```

The inverse map can be used to obtain the group element corresponding to a particular number.

```
> 6 @@ num_map;
G.1 * G.3
```

The `Random` function is sometimes useful to create a statistical profile of a group. To demonstrate, we take two groups of order 3^6 from the `SmallGroup` database.

```
> G1 := SmallGroup(3^6, 60);
> G2 := SmallGroup(3^6, 392);
```

We want to build a histogram of element orders for each group. Since these are 3-groups, each order will be a power of 3 and we use `Ilog` to get the exponent of the order. First, we define a short function to compute the histogram.

```
> function hist(G, trials)
>   // Given a 3-group G, of exponent <= 3^5,
>   // return a sequence whose ith term is the
>   // number of elements of order p^(i-1) out
>   // of trials randomly chosen elements.
>   table := [0,0,0,0,0,0];
>   for i := 1 to trials do
```

```
>     x := Random(G);
>     n := Ilog(3, Order(x));
>     table[n+1] +:= 1;
>   end for;
>   return table;
> end function;
```

Now, we use this function to compute order distributions for 100 elements in each group.

```
> t1 := hist(G1,100);
> t1;
[ 0, 0, 5, 28, 67, 0 ]
> t2 := hist(G2,100);
> t2;
[ 0, 5, 5, 25, 65, 0 ]
```

We can even display them with simple character graphics.

```
> for e in t1 do print ":","@"^e; end for;
:
:
: @@@@@
: @@@@@@@@@@@@@@@@@@@@@@@@@@@@
: @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
:
> for e in t1 do print ":","@"^e; end for;
:
: @@@@@
: @@@@@
: @@@@@@@@@@@@@@@@@@@@@@@@@
: @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
:
```

Example H63E11

Given the subgroups H and K of G, construct the set product of the groups H and K.

```
> set_product := func<G, H, K | { G | x * y : x in H, y in K }>;
```

Given a subgroup H of the pc-group G, construct H as a set of elements of G.

```
> elements := func<G, H | { G | x : x in H }>;
```

63.7 Conjugacy

`Class(H, g)`

`Conjugates(H, g)`

`g ^ H`

Given a group H and an element g belonging to a group K such that H and K are subgroups of some covering group, this function returns the set of conjugates of g under the action of H. If $H = K$, the function returns the conjugacy class of g in H.

`ConjugacyClasses(G)`

`Classes(G)`

Construct a set of representatives for the conjugacy classes of G. The classes are returned as a sequence of tuples containing the order of the elements in the class, the class length and a representative element for the class. For non-p-groups, the classes are computed using the homomorphism principle down a series with elementary abelian factors and orbit-stabilizer in each quotient. See [MN89] for details. For p-groups an algorithm based on linear algebra developed by Charles Leedham-Green is used.

`ClassMap(G)`

The class map $M : G \to \{1, \ldots, n\}$ for the group G, where n is the number of conjugacy classes of G.

`ClassRepresentative(G, x)`

The designated representative for the conjugacy class of G containing the element x (relative to existing conjugacy classes).

`IsConjugate(G, g, h)`

Given a group G and elements g and h belonging to G, return the value true if g and h are conjugate in G. The function also returns a second value in the event that the elements are conjugate: an element z which conjugates g into h.

`NumberOfClasses(G)`

`Nclasses(G)`

The number of conjugacy classes of elements of the group G.

`PowerMap(G)`

The power map M associated with the conjugacy classes of G. The map M describes where the elements of the conjugacy classes of G move under powers. That is, $< c, n >@M$ returns the class number where class c moves under the power n. The value of c must be in the range $[1 \ldots$`Nclasses(G)`$]$.

$M : \{1 \ldots n\} \times \mathbf{Z} \to \{1 \ldots n\}$

Example H63E12

Let G be a pc-representation of SL(2,3). We can compute the conjugacy classes of G. Notice that the conjugacy class object has a special printing routine, but you can still access individual entries.

```
> G := PCGroup(SpecialLinearGroup(2,GF(3)));
> G;
GrpPC : G of order 24 = 2^3 * 3
PC-Relations:
    G.1^3 = Id(G),
    G.2^2 = G.4,
    G.3^2 = G.4,
    G.4^2 = Id(G),
    G.2^G.1 = G.3 * G.4,
    G.3^G.1 = G.2 * G.3 * G.4,
    G.3^G.2 = G.3 * G.4
> Nclasses(G);
7
> cc := Classes(G);
> cc;
Conjugacy Classes of group G
----------------------------
[1]     Order 1       Length 1
        Rep Id(G)
[2]     Order 2       Length 1
        Rep G.4
[3]     Order 3       Length 4
        Rep G.1
[4]     Order 3       Length 4
        Rep G.1^2
[5]     Order 4       Length 6
        Rep G.2
[6]     Order 6       Length 4
        Rep G.1 * G.4
[7]     Order 6       Length 4
        Rep G.1^2 * G.4
> cc[3];
<3, 4, G.1>
> x := cc[3][3];
> Class(G,x);
{ G.1 * G.2 * G.3 * G.4, G.1 * G.2 * G.4, G.1, G.1 * G.3 }
7
>
```

We can use the `ClassMap` function to compute class multiplication constants (structure constants for the center of the group algebra). For example, we compute the decomposition of class 3 times class 5.

```
> cm := ClassMap(G);
```

```
> cm(G.1);
3
> i := 3; j := 5;
> t := [0: c in cc];
> for x in Class(G,cc[i][3]), y in Class(G,cc[j][3]) do
>   t[cm(x*y)] +:= 1;
> end for;
> t;
[ 0, 0, 12, 0, 0, 12, 0 ]
```

To get the actual structure constants, we need to divide each entry in `t` by the corresponding class size.

```
> [ t[i]/cc[i][2]: i in [1..#t] ];
[ 0, 0, 3, 0, 0, 3, 0 ]
```

63.8 Subgroups

Subgroups of pc-groups are treated as independent pc-groups in their own right, with the subgroup relationship maintained in internal data structures. Thus, a subgroup has generators and a pc-presentation and one can apply any of the functions described earlier for groups. Furthermore, there are a variety of functions and operations specifically involving subgroups.

63.8.1 Definition of Subgroups by Generators

The most flexible method of defining a subgroup is to list generators or normal generators for the subgroup.

`sub< G | L >`

Construct the subgroup H of the pc-group G generated by the elements specified by the terms of the *generator list* L.

A term $L[i]$ of the generator list may consist of any of the following objects:

(a) An element liftable to G (in particular, any element of G);

(b) A subgroup of G;

(c) A set or sequence of (a), or (b).

The collection of words and groups specified by the list must all belong to the group G and H will be constructed as a subgroup of G.

The subgroup H is defined to be generated by the words specified directly by terms $L[i]$ together with the stored generating words for any groups specified by terms of $L[i]$. MAGMA will compute a set of pc-generators for H and, if H is a p-group, a minimal generating set.

The inclusion map from H to G is returned as well.

`ncl< G | L >`

Construct the subgroup N of the pc-group G as the normal closure of the subgroup generated by the elements specified by the terms of the *generator list* L.

The possible forms of a term $L[i]$ of the generator list are the same as for the `sub`-constructor.

The inclusion map from N to G is returned as well.

Example H63E13

We define G to be Z_5 wr Z_3 and then create two subgroups. Notice that the `ncl`-constructor builds a larger subgroup in this case.

```
> G<a,b,c,d> := PolycyclicGroup<a,b,c,d| a^3, b^5, c^5, d^5,
>   b^a = c, c^a = d, d^a = b>;
> H := sub<G| b,c>;
> Order(H);
25
> IsAbelian(H);
true
> IsNormal(G, H);
false
> N := ncl<G| b,c>;
> IsNormal(G, N);
true
> Order(N);
125
```

63.8.2 Membership and Coercion

There are several functions and operators which allow one to take advantage of the subgroup relationship to rewrite elements from one presentation to another. That is, if x is an element of H which is a subgroup of G, then x has a representation as a normal word in the pc-generators of H, but also has a representation as a (different) normal word in the pc-generators of G. The coercion operator and inclusion map allow one to compute one of these words based on the other, thus shifting where we view the element in question.

MAGMA keeps track of the various relationships between subgroups in a group. Thus, if H is a subgroup of K which is a subgroup of G, then H can also be considered a subgroup G. Similarly, in situations involving elements of two groups, A and B, MAGMA will often try to find a *covering group* C which contains both of A and B. In this case, the elements may be automatically coerced into the covering group.

`g in G`

Given an element g and a group G, return `true` if g is an element of G, `false` otherwise. In order for this comparison to make sense, both g and G must be contained in some covering group.

`g notin G`

Given an element g and a group G, return `true` if g is not an element of G, `false` otherwise. In order for this comparison to make sense, both g and G must be contained in some covering group.

`G ! g`

Given an element g belonging to some subgroup H of the group G, rewrite g as an element of G.

`H ! g`

Given an element g belonging to the group G, and given a subgroup H of G containing g, rewrite g as an element of H.

`K ! g`

Given an element g belonging to the group H, and a group K, such that H and K are subgroups of a covering group G, and both H and K contain g, rewrite g as an element of K.

Example H63E14

We create two subgroups of a dihedral group.

```
> G := DihedralGroup(GrpPC, 10);
> C := sub<G| G.2>;
> H := sub<G| G.1, G.3>;
> H.1 in C;
false
> H.2 in C;
true
> Parent(H.1);
GrpPC : H of order 10 = 2 * 5
PC-Relations:
    H.1^2 = Id(H),
    H.2^5 = Id(H),
    H.2^H.1 = H.2^4
> G!(H.1);
G.1
```

MAGMA will compute appropriate covering groups as needed.

```
> H.1*C.1;
G.1 * G.2 * G.3^2
> x := (H.1, C.2);
> x;
G.3^2
> H!x;
H.2^2
> C!x;
```

```
C.2^2
> C!(H.2);
C.2
```

63.8.3 Inclusion and Equality

`S subset G`

Given an group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G`

Given a group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is not a subset of G, `false` otherwise.

`H subset G`

Given groups G and H, subgroups of some covering group, return `true` if H is a subgroup of G, `false` otherwise.

`H notsubset G`

Given groups G and H, subgroups of some covering group, return `true` if H is not a subgroup of G, `false` otherwise.

`G eq H`

Given groups G and H, subgroups of some covering group, return `true` if G and H are the same group, `false` otherwise.

`G ne H`

Given groups G and H, subgroups of some covering group, return `true` if G and H are distinct groups, `false` otherwise.

`InclusionMap(G, H)`

The map from the subgroup H of G to G.

63.8.4 Standard Subgroup Constructions

The operators and functions which construct a subgroup of a pc-group always return the subgroup as a pc-group.

```
H ^ g
```

```
Conjugate(H, g)
```

Construct the conjugate $g^{-1} * H * g$ of the group H under the action of the element g. The group H and the element g must belong to a common group.

```
H meet K
```

The intersection of groups H and K. The algorithm used for non-p-groups is described in [GS90].

```
H meet:= K
```

Replace H with the intersection of groups H and K.

```
CommutatorSubgroup(G, H, K)
```

```
CommutatorSubgroup(H, K)
```

Construct the commutator subgroup of groups H and K, where H and K are subgroups of a common group G.

```
Centralizer(G, g)
```

```
Centraliser(G, g)
```

The centralizer of the element g in the group G.

```
Centralizer(G, H)
```

```
Centraliser(G, H)
```

The centralizer of the subgroup H in the group G.

```
Core(G, H)
```

The maximal normal subgroup of G that is contained in the subgroup H of G.

```
H ^ G
```

```
NormalClosure(G, H)
```

The normal closure of the subgroup H in the group G.

```
Normalizer(G, H)
```

```
Normaliser(G, H)
```

The normalizer of the subgroup H of the group G. The algorithm used for non-p-groups is described in [GS90].

Example H63E15

We'll consider various subgroups of a direct product of a cyclic group of order 6 and dihedral group of order 10.

```
> G := DirectProduct(CyclicGroup(GrpPC,6), DihedralGroup(GrpPC,5));
> x := G.3;
> C := Centralizer(G,x);
> C;
GrpPC : C of order 12 = 2^2 * 3
PC-Relations:
    C.1^2 = Id(C),
    C.2^2 = Id(C),
    C.3^3 = Id(C)
> H := sub<G|G.2,G.4>;
> Order(H);
15
```

We can compute the intersection using the `meet` operator.

```
> K := H meet C;
> K;
GrpPC : K of order 3
PC-Relations:
    K.1^3 = Id(K)
```

To get the join of two subgroups, we simply use the `sub`-constructor.

```
> J := sub<G|H, C>;
> J eq G;
true
```

63.8.5 Properties of Subgroups

`Index(G, H)`

The index of the subgroup H in the group G, returned as an ordinary integer.

`FactoredIndex(G, H)`

The factored index of the subgroup H in the group G.

63.8.6 Predicates for Subgroups

`IsCentral(G, H)`

Returns `true` if the subgroup H of the group G lies in the centre of G, `false` otherwise.

`IsConjugate(G, H, K)`

Given a group G and subgroups H and K belonging to G, return the value `true` if H and K are conjugate in G. The function returns a second value in the event that the subgroups are conjugate: an element z which conjugates H into K.

`IsMaximal(G, H)`

Returns `true` if the subgroup H of the group G is a maximal subgroup of G, `false` otherwise.

`IsNormal(G, H)`

Returns `true` if the subgroup H of the group G is a normal subgroup of G, `false` otherwise.

`IsSelfNormalizing(G, H)`

Returns `true` if the subgroup H of the group G is self-normalizing in G, `false` otherwise.

`IsSubnormal(G, H)`

Returns `true` if the subgroup H of the group G is subnormal in G, `false` otherwise.

Example H63E16

```
> G := PCGroup(Sym(4));
> G;
GrpPC : G of order 24 = 2^3 * 3
PC-Relations:
    G.1^2 = Id(G),
    G.2^3 = Id(G),
    G.3^2 = Id(G),
    G.4^2 = Id(G),
    G.2^G.1 = G.2^2,
    G.3^G.1 = G.3 * G.4,
    G.3^G.2 = G.4,
    G.4^G.2 = G.3 * G.4
> U := sub<G|G.4>;
> IsNormal(G,U);
false
> IsSubnormal(G,U);
```

```
true
```

Now, we try to construct a subnormal chain by taking normalizers.

```
> N1 := Normalizer(G,U);
> Index(G,N1);
3
> N2 := Normalizer(G,N1);
> Index(G,N2);
3
> N1 eq N2;
true
```

We're stuck. However, we can work our way down with `NormalClosure`.

```
> M1 := NormalClosure(G,U);
> U subset M1;
true
> M1 subset U;
false
> M2 := NormalClosure(M1,U);
> M2 eq U;
true
```

Now, we work inside the Sylow 2-subgroup and look for complements of the cyclic group of order 4 by brute force.

```
> S := Sylow(G,2);
> S;
GrpPC : S of order 8 = 2^3
PC-Relations:
    S.2^S.1 = S.2 * S.3
> T := sub<S|S.1*S.2>;
> list := [];
> for x in S do
>   if (Order(x) ne 2) or (x in T) then
>     continue;
>   end if;
>   Append(~list, sub<S|x>);
> end for;
> #list;
4
> for i in [1..3], j in [i+1..4] do
>   print i,j,IsConjugate(S,list[i],list[j]);
> end for;
1 2 true S.1
1 3 false
1 4 false
2 3 false
2 4 false
```

```
3 4 true S.2
```

We see that T has two conjugacy classes of complements.

63.8.7 Hall π-Subgroups and Sylow Systems

The functions given here all assume that G is a soluble group having order $p_1^{e_1}p_2^{e_2}\cdots p_k^{e_k}$.

`ComplementBasis(G)`

A complement basis of the soluble group G. This is a sequence of k subgroups of G, where the i-th subgroup has order $p_1^{e_1}\ldots p_{i-1}^{e_{i-1}}p_{i+1}^{e_{i+1}}\ldots p_k^{e_k}$, i.e. the complements of the k Sylow subgroups of G.

`HallSubgroup(G, S)`

The Hall π-subgroup of G, where π is defined by S. The argument S may be a set of primes, a single prime, or the negation of a single prime. If $S = -p$, then the Hall p'-subgroup of G is returned.

`pCore(G, S)`

The core of the Hall π-subgroup, where π is defined by the argument S, which has the same interpretation as for `HallSubgroup`.

`SylowBasis(G)`

A Sylow basis for the soluble group G. This is a sequence of k subgroups of G, having orders $p_1^{e_1},\ldots,p_k^{e_k}$, i.e. the k Sylow subgroups of G.

`SylowSubgroup(G, p)`

`Sylow(G, p)`

A Sylow p-subgroup for the group G.

`SystemNormalizer(G)`

`SystemNormaliser(G)`

The system normalizer for the group G. The system normalizer of the complement basis $\Sigma = \{H_1, ..., H_k\}$ is defined to be the intersection of the normalizers in G of each H_i, ie. $N(\Sigma) = \cap_{i=1}^{k} N_G(H_i)$. The algorithm used is derived directly from the definition.

Example H63E17

Given the group $D_3 \wr D_5$, we can construct the Hall 2-subgroup as follows:

```
> H := DihedralGroup(GrpPerm, 5);
> G := WreathProduct(DihedralGroup(GrpPC, 3), DihedralGroup(GrpPC, 5),
>         [H.2, H.1]);
> H2 := HallSubgroup(G, 2);
> Order(H2);
64
```

The Hall $2'$-subgroup of the same group is constructed as follows:

```
> H35 := HallSubgroup(G, -2);
> Order(H35);
1215
```

63.8.8 Conjugacy Classes of Subgroups

MAGMA has functions for computing the subgroups of a group that return the subgroups either as a list of conjugacy class representatives as or as a poset. Details of these functions may be found in Chapter 57. Here we mention the basic functions for convenience.

`SubgroupClasses(G)`

`Subgroups(G)`

Conjugacy class representatives for all subgroups of G. The algorithm was developed by M. Slattery and is essentially that of [Hul99] without the action of automorphisms.

`AbelianSubgroups(G)`

`CyclicSubgroups(G)`

`ElementaryAbelianSubgroups(G)`

`NilpotentSubgroups(G)`

Conjugacy class representatives for all subgroups of the indicated type in G. The algorithm used is essentially that of [Hul99] without the action of automorphisms. Appropriate filters are applied to select the desired groups at each successive quotient in the computation.

`MaximalSubgroups(G)`

A sequence of conjugacy class representatives for the maximal subgroups of G. The algorithm, developed by Charles Leedham-Green, relies on computing a special presentation for G.

`SubgroupLattice(G)`

The lattice of conjugacy classes of subgroups of G.

`BurnsideMatrix(G)`

The Burnside matrix corresponding to the lattice of subgroups of G.

`DisplayBurnsideMatrix(G)`

Pretty-print the Burnside matrix corresponding to the lattice of subgroups of G.

Example H63E18

To show a bit about subgroup classes, we look at the direct product of C_3 and D_3. First we list out the normal subgroups of G.

```
> G := DirectProduct(CyclicGroup(GrpPC,3),
>                    DihedralGroup(GrpPC,3));
> ns := NormalSubgroups(G);
> ns;
Conjugacy classes of subgroups
------------------------------
[1]     Order 1            Length 1
        GrpPC of order 1
        PC-Relations:
[2]     Order 3            Length 1
        GrpPC of order 3
        PC-Relations:
            $.1^3 = Id($)
[3]     Order 3            Length 1
        GrpPC of order 3
        PC-Relations:
            $.1^3 = Id($)
[4]     Order 6            Length 1
        GrpPC of order 6 = 2 * 3
        PC-Relations:
            $.1^2 = Id($),
            $.2^3 = Id($),
            $.2^$.1 = $.2^2
[5]     Order 9            Length 1
        GrpPC of order 9 = 3^2
        PC-Relations:
            $.1^3 = Id($),
            $.2^3 = Id($)
[6]     Order 18           Length 1
        GrpPC of order 18 = 2 * 3^2
        PC-Relations:
            $.1^2 = Id($),
            $.2^3 = Id($),
            $.3^3 = Id($),
```

```
            $.3^$.1 = $.3^2
```

The normal subgroups sequence has a special printing routine. Each entry in the sequence is actually a record.

```
> ns[2];
rec<recformat<order, length, subgroup, presentation> |
 order := 3, length := 1, subgroup := GrpPC of order 3
PC-Relations:
    $.1^3 = Id($)>
```

We extract the two normal subgroups of order 3. Each of them turn out to have one conjugacy class of complements in G. However, one of the complements is normal and the other is not.

```
> N1 := ns[2]`subgroup;
> N2 := ns[3]`subgroup;
> c1 := Complements(G,N1);
> c1;
[
    GrpPC of order 6 = 2 * 3
    PC-Relations:
        $.1^2 = Id($),
        $.2^3 = Id($),
        $.2^$.1 = $.2^2
]
> c2 := Complements(G,N2);
> c2;
[
    GrpPC of order 6 = 2 * 3
    PC-Relations:
        $.1^2 = Id($),
        $.2^3 = Id($)
]
> Index(G,Normalizer(G,c1[1]));
1
> Index(G,Normalizer(G,c2[1]));
3
```

We can look at the full list of classes of subgroups of G to see that there are three classes of non-normal subgroups as well as the normal subgroups. There are two non-normal subgroups of order 3 in addition to $N1$ and $N2$.

```
> Subgroups(G);
Conjugacy classes of subgroups
------------------------------
[ 1]    Order 1            Length 1
        GrpPC of order 1
        PC-Relations:
[ 2]    Order 2            Length 3
        GrpPC of order 2
        PC-Relations:
```

```
          $.1^2 = Id($)
[ 3]    Order 3           Length 1
        GrpPC of order 3
        PC-Relations:
          $.1^3 = Id($)
[ 4]    Order 3           Length 1
        GrpPC of order 3
        PC-Relations:
          $.1^3 = Id($)
[ 5]    Order 3           Length 2
        GrpPC of order 3
        PC-Relations:
          $.1^3 = Id($)
[ 6]    Order 6           Length 1
        GrpPC of order 6 = 2 * 3
        PC-Relations:
          $.1^2 = Id($),
          $.2^3 = Id($),
          $.2^$.1 = $.2^2
[ 7]    Order 6           Length 3
        GrpPC of order 6 = 2 * 3
        PC-Relations:
          $.1^2 = Id($),
          $.2^3 = Id($)
[ 8]    Order 9           Length 1
        GrpPC of order 9 = 3^2
        PC-Relations:
          $.1^3 = Id($),
          $.2^3 = Id($)
[ 9]    Order 18          Length 1
        GrpPC of order 18 = 2 * 3^2
        PC-Relations:
          $.1^2 = Id($),
          $.2^3 = Id($),
          $.3^3 = Id($),
          $.3^$.1 = $.3^2
```

63.9 Quotient Groups

63.9.1 Construction of Quotient Groups

One of the strengths of representing groups with polycyclic or power-conjugate presentations is that arbitrary quotient groups can be computed. Given (generators for) a normal subgroup of a pc-group, MAGMA will compute a pc-presentation for the quotient and the corresponding canonical homomorphism.

The `pQuotient` function, which can be used to find a prime-power quotient of a finitely-presented group, can also be used to compute quotients of pc-groups.

`quo< G | L >`

Construct the quotient Q of the pc-group G by the normal subgroup N, where N is the smallest normal subgroup of G containing the elements specified by the terms of the *generator list* L.

The possible forms of a term $L[i]$ of the generator list are the same as for the `sub`-constructor.

The quotient group Q and the corresponding natural homomorphism $f : G \to Q$ are returned.

`G / N`

Given a normal subgroup N of the pc-group G, construct the quotient of G by N.

Example H63E19

We will compute $O_{3',3}(G)$, where G is a pc-representation of the symmetric group S_4. The subgroup is defined by $O_{3',3}(G)/O_{3'}(G) = O_3(G/O_{3'}(G))$.

```
> G := PCGroup(Sym(4));
> N := pCore(G,-3);
> Q,f := quo<G|N>;
> Q;
GrpPC : Q of order 6 = 2 * 3
PC-Relations:
    Q.1^2 = Id(Q),
    Q.2^3 = Id(Q),
    Q.2^Q.1 = Q.2^2
> S := pCore(Q,3);
> H := S @@ f;
> H;
GrpPC : H of order 12 = 2^2 * 3
PC-Relations:
    H.1^3 = Id(H),
    H.2^2 = Id(H),
    H.3^2 = Id(H),
    H.2^H.1 = H.2 * H.3,
    H.3^H.1 = H.2
```

63.9.2 Abelian and p-Quotients

A number of standard quotients may be constructed.

`AbelianQuotient(G)`

The maximal abelian quotient G/G' of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism $\pi : G \to G/G'$ is returned as second value.

`AbelianQuotientInvariants(G)`

`AQInvariants(G)`

A sequence of integers giving the abelian invariants of the maximal abelian quotient of G.

`ElementaryAbelianQuotient(G, p)`

The maximal p-elementary abelian quotient Q of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism $\pi : G \to Q$ is returned as second value.

`pQuotient(G, p, c : parameters)`

`Workspace`	RngIntElt	*Default :* 5000000
`Metabelian`	BoolElt	*Default :* `false`
`Exponent`	RngIntElt	*Default :* 0
`Print`	RngIntElt	*Default :* 0

Given a pc-group G, a prime p, and a positive integer c, this function constructs a consistent power-conjugate presentation for the largest p-quotient P of G having lower exponent-p class at most c. If c is given as zero, then the limit 127 is placed on the class.

The function also returns the natural homomorphism π from G to P, a sequence S describing the definitions of the pc-generators of P and a flag indicating whether P is the maximal p-quotient of G.

The k-th element of S is a sequence of two integers, describing the definition of the k-th pc-generator $P.k$ of P as follows.

- If $S[k] = [0, r]$, then $P.k$ is defined via the image of $G.r$ under π.
- If $S[k] = [r, 0]$, then $P.k$ is defined via the power relation for $P.r$.
- If $S[k] = [r, s]$, then $P.k$ is defined via the conjugate relation involving $P.r^{P.s}$.

63.10 Normal Subgroups and Subgroup Series

63.10.1 Characteristic Subgroups

`Centre(G)`
`Center(G)`

The centre of the group G.

`CommutatorSubgroup(G)`
`DerivedSubgroup(G)`
`DerivedGroup(G)`

The derived subgroup of the group G.

`FittingSubgroup(G)`
`FittingGroup(G)`

The Fitting subgroup of the group G.

`FrattiniSubgroup(G)`

The Frattini subgroup of the group G.

`Hypercentre(G)`
`Hypercenter(G)`

The hypercentre of the group G, i.e. the stationary term in the upper central series for G.

`MinimalNormalSubgroups(G)`

A sequence containing all minimal normal subgroups of G.

`pCore(G, S)`

The maximal normal π-subgroup of G, $O_\pi(G)$, where π is defined by S. The argument S may be a set of primes, a single prime, or the negation of a single prime. If $S = -p$, then $O_{p'}(G)$ is returned.

`Socle(G)`

The socle of G.

63.10.2 Subgroup Series

`AbelianBasis(G)`

Given an abelian group G, return sequences B and I such that order$(B[i]) = I[i]$ and $\langle B \rangle = G$ and the terms of I give the types of each p-primary component of G.

`AbelianInvariants(G)`

`Invariants(G)`

The abelian invariants of the abelian group G as a sequence of integers.

`ChiefSeries(G)`

A chief series for the group G. The series is returned as a sequence of subgroups of G.

`CompositionSeries(G)`

A composition series for the group G. The series is returned as a sequence of subgroups of G. The i-th term of the composition series has a presentation given by the generators $G.i$ through $G.$`NPCgens`(G) and relations involving those generators only.

`CompositionFactors(G)`

A sequence of integer tuples that describe the composition factors, ordered according to some composition series for the group G. Since each factor will be a cyclic group of prime order, the tuples will each be of the form $< 19, 0, q >$ representing the cyclic group of order q. The sequence has a custom print routine.

`CompositionSeries(G, i)`

The $i+1$-th entry of the composition series for the group G. Its presentation is given by the generators $G.(i+1)$ through $G.m$, where m is the number of pc-generators of G and relations involving these generators only.

`DerivedSeries(G)`

The derived series of the group G. The series is returned as a sequence of subgroups.

`DerivedLength(G)`

The derived length of the group G.

`ElementaryAbelianSeries(G)`

An elementary abelian series is a chain of normal subgroups with the property that the quotient of each pair of successive terms in the series is elementary abelian. The elementary abelian series for the group G is returned as a sequence of subgroups.

`ElementaryAbelianSeriesCanonical(G)`

Gives a similar result to using `ElementaryAbelianSeries`, except the series returned depends only on the isomorphism type of the group, and consists of characteristic subgroups. This function may be slower than `ElementaryAbelianSeries`.

`LowerCentralSeries(G)`

The lower central series for the group G. The series is returned as a sequence of subgroups.

`NilpotencyClass(G)`

If G is nilpotent, return the nilpotence class of G. Otherwise, -1 is returned.

`pCentralSeries(G, p)`

The p-central series for G, where p is a prime dividing $|G|$. The series is returned as a sequence of subgroups. The p-central series $P_1 \triangleright P_2 \triangleright \cdots \triangleright P_i$ of a soluble group G is defined inductively as follows:

$P_1 = G$,

$P_{i+1} = (G, P_i)P_i^p$, for $i > 0$.

`SubnormalSeries(G, H)`

Given a group G and a subgroup H of G, return a sequence of subgroups commencing with G and terminating with H, such that each subgroup is normal in the previous one. If H is not subnormal in G, the empty sequence is returned.

`UpperCentralSeries(G)`

The upper central series of G. The series is returned as a sequence of subgroups.

Example H63E20

The elementary abelian series of the group $D_3 \wr D_5$ has terms of the following orders:

```
> H := DihedralGroup(GrpPerm, 5);
> G := WreathProduct(DihedralGroup(GrpPC, 3), DihedralGroup(GrpPC, 5),
>       [H.2, H.1]);
> EAS := ElementaryAbelianSeries(G);
> for i := 1 to #EAS do
>    print FactoredOrder(EAS[i]);
> end for;
    [ <2, 6>, <3, 5>, <5, 1> ]
    [ <2, 4>, <3, 5>, <5, 1> ]
    [ <2, 4>, <3, 5> ]
    [ <3, 5> ]
    []
```

Hence the elementary abelian factors can be seen to have sizes 2^2, 5, 2^4, and 3^5, reading from top to bottom.

63.10.3 Series for p-groups

The following functions are only defined for a pc-group which is a p-group.

`Agemo(G, i)`

Given a p-group G, return the characteristic subgroup of G generated by the elements x^{p^i}, $x \in G$, where i is a positive integer.

`Omega(G, i)`

Given a p-group G, return the characteristic subgroup of G generated by the elements of order dividing p^i, where i is a positive integer.

`JenningsSeries(G)`

Given a p-group G, return the Jennings series for G. The series is returned as a sequence of subgroups. The Jennings series $J_1 \triangleright J_2 \triangleright \cdots \triangleright J_i \cdots$ of a p-group G is defined inductively as follows:

$J_1 = G$,

$J_{i+1} =< (J_i, G), J_k^p >$, with $k = \lceil (i+1)/p \rceil$, $i > 0$.

`pClass(G)`

The lower exponent-p class of the p-group G.

`pRanks(G)`

A sequence whose i-th entry is the number of pc-generators for the lower exponent-p class i quotient of the p-group G.

63.10.4 Normal Subgroups and Complements

`NormalSubgroups(G)`

The collection of all normal subgroups of G returned as a sequence.

`NormalLattice(G)`

The lattice of normal subgroups of G.

`MinimalNormalSubgroup(G)`

An elementary abelian minimal normal subgroup of the soluble group G.

`MinimalNormalSubgroup(G, N)`

Given a non-trivial, normal subgroup N of G, return an elementary abelian minimal normal subgroup of G contained in N.

`Complements(G, N)`

Given a normal subgroup N of G, return conjugacy class representatives of all complements of N in G. This function implements the first cohomology computation described in [CNW90].

`NormalComplements(G, N)`

Given a normal subgroup N of G, return all normal complements of N in G. This function implements the first cohomology computation described in [CNW90].

`NormalComplements(G, H, N)`

Given a normal subgroup N of G, and a normal subgroup H of G containing N, return all complements of N in H which are normal in G. This function implements the first cohomology computation described in [CNW90].

Example H63E21

We define the direct product of an extraspecial group of order 3^3 and D_3 and let N be the first factor of this product. Inside the Sylow 3-subgroup, we see that N has 11 classes of complements, three of which are normal.

```
> A := ExtraSpecialGroup(GrpPC,3,1);
> B := DihedralGroup(GrpPC,3);
> G,f,p := DirectProduct(A,B);
> N := f[1](A);
> S3 := Sylow(G,3);
> cS := Complements(S3,N);
> [Index(S3,Normalizer(S3,t)):t in cS];
[ 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3 ]
```

We can compute only the normal complements by using `NormalComplements`.

```
> ncS := NormalComplements(S3,N);
> #ncS;
3
```

We can check that precisely one of these three complements is actually normal in G.

```
> [IsNormal(G,t):t in ncS];
[ true, false, false ]
```

Since N has a G-normal complement in $S3$, we must have $S3$ normal in G. We can verify this. Using the three-parameter version of `NormalComplements` we can directly compute the G-normal complements of N in $S3$.

```
> IsNormal(G,S3);
true
> ncG := NormalComplements(G,S3,N);
> #ncG;
1
> #NormalComplements(G,N);
```

```
1
```

63.11 Cosets

63.11.1 Coset Tables and Transversals

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a group G and a subgroup H of G, this function returns

(a) An indexed set of elements T of G forming a right transversal for G over H; and

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and g in G, ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

`CosetTable(G, H)`

Given a group G and a subgroup H of G of index r, return a mapping $M : \langle\{1..r\}, G\rangle \to \{1..r\}$ describing the action of G on the (right) cosets of H.

`Transversal(G, H, K)`

An indexed set of representatives for the double cosets HuK in G, and the corresponding transversal mapping. The algorithm used is described in [Sla01].

`ShortCosets(p, H, G)`

Computes a set of representatives for the transversal of G modulo H of all cosets that contain p. This computation does not do a full transversal of G modulo H and may therefore be used even if the index of $(G : H)$ is very large.

63.11.2 Action on a Coset Space

`CosetAction(G, H)`

Given a subgroup H of the group G, construct the permutation representation of G given by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The natural homomorphism $f : G \to L$;

(b) The induced group L;

(c) The kernel K of the action (a subgroup of G).

`CosetImage(G, H)`

Given a subgroup H of the group G, construct the image L of G given by the action of G on the set of (right) cosets of H in G. L is returned as a permutation group.

`CosetKernel(G, H)`

Given a subgroup H of the group G, construct the kernel of the action of G on the set of (right) cosets of H in G.

63.12 Automorphism Group

63.12.1 General Soluble Group

In the case of a soluble non-p-group there are two algorithms available. By default, a lifting-based algorithm developed by M. Smith [Smi94] and extended by Smith and Slattery to use second cohomology is used. Alternatively, there is also an algorithm developed by D. Howden [How12], which uses the automorphism group of a Sylow p-subgroup of G to construct the automorphism group of G.

63.12.1.1 Lifting Algorithm

The automorphism group is computed step-by-step considering a series of factors of G by terms of a characteristic series

$$G = G_1 > G_2 > \cdots > G_k > 1$$

such that G_i/G_{i+1} is elementary abelian for each i. We compute the automorphism group $\mathrm{Aut}(G/G_2) \cong \mathrm{GL}(d, p)$ for some integer d and prime p, and then lift through each of the elementary abelian layers $G/G_3, \ldots, G/G_{k-1}$, finally arriving at $\mathrm{Aut}(G/G_k) = \mathrm{Aut}(G)$.

A group of type `GrpAuto` is returned. Details of the computation can be seen by setting the verbose flag `AutomorphismGroup` to `true`, and the characteristic series is available as the attribute `CharacteristicSeries` on the returned group.

In addition to the usual properties of `GrpAuto` (such as `Order`, `Ngens`, etc.), two special fields, `GenWeights` and `WeightSubgroupOrders`, are provided for automorphism groups of (non p-group) pc-groups. These each relate to weight subgroups of the automorphism group. Let i be the largest subscript such that the automorphism acts trivially on G/G_i. Then the automorphism is said to have weight $2i + 1$ if it acts non-trivially on G_i/G_{i+1}, and weight $2i + 2$ if it acts trivially on G_i/G_{i+1}. Note that there are no automorphisms of weight 2. The automorphisms of weight greater than or equal to a given value form a normal subgroup of A.

`AutomorphismGroup(G)`

Given a soluble group G presented by a pc-presentation, this function returns the automorphism group of G as a group of type `GrpAuto`.

`HasAttribute(A, "GenWeights")`

If the attribute `GenWeights` is defined for A then the function returns `true`, and a sequence of integers. This integer sequence indicates where each generator lies in the normal series of A corresponding to the action of the group on G (as described at the beginning of the section). If the attribute is not set then the sequence is unassigned. The function call `AutomorphismGroup(G)`, where G has type `GrpPC`, always returns an automorphism group with this attribute set. In this case the sequence may also be obtained by the short form `A‘GenWeights`.

```
HasAttribute(A, "WeightSubgroupOrders")
```

If the attribute `WeightSubgroupOrders` is defined for A then the function returns `true`, and a sequence of integers. This sequence of integers gives the orders for the normal series of weight subgroups described at the beginning of the section. If the attribute is not set then the sequence is unassigned. The function call `AutomorphismGroup(G)`, where G has type `GrpPC`, always returns an automorphism group with this attribute set. In this case the sequence may also be obtained by the short form `A‘WeightSubgroupOrders`.

Example H63E22

An example using `AutomorphismGroup` and some related features. We build a group based on the structure of a finite field (multiplicative group acting on the additive group) and then compute its automorphism group. First, we set up the field.

```
> E := GF(2);
> F := GF(8);
> V,phi := VectorSpace(F,E);
> d := Dimension(V);
> x := PrimitiveElement(F);
```

Then, define a pc-group to act and define the action based on the multiplication in the field. Compute the matrix by mapping the vectors back to the field, multiplying by x, and then recording the result.

```
> C := CyclicGroup(GrpPC,Order(x));
> MR := MatrixRing(E, d);
> s := [];
> for i := 1 to d do
>  y := ((V.i)@@phi)*x;
>  s cat:= Eltseq(y);
> end for;
```

Turn the sequence of image components into a matrix and use the matrix to create a C-module. Then use that module to create the split extension.

```
> t := MR!s;
> M := GModule(C,[t]);
> G := Extension(M,C);
> G;
GrpPC : G of order 56 = 2^3 * 7
PC-Relations:
    G.1^7 = Id(G),
    G.2^2 = Id(G),
    G.3^2 = Id(G),
    G.4^2 = Id(G),
    G.2^G.1 = G.3,
    G.3^G.1 = G.4,
```

```
    G.4^G.1 = G.2 * G.3
```

Then we can compute the automorphism group of G.

```
> A := AutomorphismGroup(G);
> A;
A group of automorphisms of GrpPC : G
Generators:
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1^2
        G.2 |--> G.3 * G.4
        G.3 |--> G.2 * G.4
        G.4 |--> G.3
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1
        G.2 |--> G.2 * G.3
        G.3 |--> G.3 * G.4
        G.4 |--> G.2 * G.3 * G.4
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1 * G.2 * G.3
        G.2 |--> G.2
        G.3 |--> G.3
        G.4 |--> G.4
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1 * G.3 * G.4
        G.2 |--> G.2
        G.3 |--> G.3
        G.4 |--> G.4
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1 * G.4
        G.2 |--> G.2
        G.3 |--> G.3
        G.4 |--> G.4
> [Order(x):x in Generators(A)];
[ 3, 2, 7, 2, 2 ]
```

Next, we can use the automorphisms to create an extension of G.

```
> b := A.1;
> Order(b);
3
> tau := hom<G->G|[b(G.i):i in [1..NPCgens(G)]]>;
> D := CyclicGroup(GrpPC,Order(b));
> K := Extension(G,D,[tau]);
> K;
GrpPC : K of order 168 = 2^3 * 3 * 7
PC-Relations:
    K.1^3 = Id(K),
    K.2^7 = Id(K),
    K.3^2 = Id(K),
```

```
    K.4^2 = Id(K),
    K.5^2 = Id(K),
    K.2^K.1 = K.2^2,
    K.3^K.1 = K.4 * K.5,
    K.3^K.2 = K.4,
    K.4^K.1 = K.3 * K.5,
    K.4^K.2 = K.5,
    K.5^K.1 = K.4,
    K.5^K.2 = K.3 * K.4
> #Classes(K);
8
```

Finally, we examine information about the weight subgroups. We list only the orders of the terms of the characteristic series in G in order to save space.

```
> [Order(H): H in A‘CharacteristicSeries];
[ 56, 8, 1 ]
> A‘GenWeights;
[ 1, 3, 4, 4, 4 ]
> A‘WeightSubgroupOrders;
[ 168, 56, 56, 8 ]
```

63.12.1.2 Lifting from the Automorphism Group of a Sylow p-subgroup

The algorithm developed by D. Howden [How12] follows a different strategy to the Smith-Slattery lifting approach. Given a soluble group G, the algorithm determines a suitable Sylow p-subgroup P of G, and uses the automorphism group of P (using the algorithm for p-groups detailed below) to construct the automorphism group of G.

In cases where the automorphism group is soluble, the algorithm automatically constructs a pc-representation for it. Solublility of the returned group can then be tested via the `IsSoluble` intrinsic. The pc-representation obtained using the `PCGroup` intrinsic, and pc-generators (as automorphism maps) via the `PCGenerators` intrinsic.

In some cases where the automorphism group is not soluble, the algorithm will construct a permutation representation during its construction.

A group of type `GrpAuto` is returned. Details of the computation can be seen by setting the verbose flag `AutomorphismGroupSolubleGroup` to 1.

A variation of this algorithm can be used for isomorphism testing.

`AutomorphismGroupSolubleGroup(G: parameters)`

Given a soluble group G presented by a pc-presentation, this function returns the automorphism group of G as a group of type GrpAuto.

p [RngIntElt] *Default :* 1;

A prime p dividing the order of G. The automorphism group of the Sylow p-subgroup is then used to construct the automorphism group of G. If the p-core of G is trivial

then an error is given. The default value of $p = 1$ indicates that the algorithm should take p to be the prime giving the largest Sylow p-subgroup of G.

`IsIsomorphicSolubleGroup(G, H: parameters)`

Returns `true` if the soluble groups G, H presented by pc-presentations, are isomorphic, and `false` otherwise. Where the groups are isomorphic, a mapping $G \to H$ is also returned.

p [RngIntElt] *Default :* 1;

A prime p dividing the order of G (assuming that the orders of G and H are the same). The algorithm then tests Sylow p-subgroups of G and H for isomorphism and attempts to extend these isomorphisms to an isomorphism $G \to H$. If the p-cores of G and H are trivial then an error is given. The default value of $p = 1$ indicates that the algorithm should take p to be the prime giving the largest Sylow p-subgroup of G.

Example H63E23

An example using `AutomorphismGroupSolubleGroup` and some related features. We use a group from the `solgps` library.

```
> load solgps;
> G := G10();
> FactoredOrder(G);
[ <2, 18>, <7, 4> ]
> time A := AutomorphismGroupSolubleGroup(G);
Time: 18.220
> time R_A, phi_A := PCGroup(A);
Time: 0.000
> FactoredOrder(R_A);
[ <2, 20>, <3, 2>, <7, 6> ]
```

63.12.2 p-group

For a description of the algorithm used to construct the automorphism group of a p-group, see [ELGO02].

While it is difficult to state very firm guidelines for the performance of the algorithm, our experience suggests that it has most difficulty in constructing automorphism groups of p-groups of "large" Frattini rank (say rank larger than about 6) and p-class 2. If the group has larger p-class, then it usually has more characteristic structure and the algorithm exploits this. The *order* of a group is *not* a useful guide to the difficulty of the computation.

`SetVerbose ("AutomorphismGroup", 1)` provides information on the progress of the algorithm.

`AutomorphismGroup(G: parameters)`

The group G is a p-group described by a pc-presentation. The function returns the automorphism group of G as a group of type `GrpAuto`.

`CharacteristicSubgroups`

[GrpPC] *Default :* [];

A list of known characteristic subgroups of G; these may improve the efficiency of the construction. Note that the algorithm simply accepts that the supplied subgroups are fixed under the action of the automorphism group; it does *not* verify that they are in fact characteristic.

Example H63E24

```
> G := SmallGroup (64, 78);
> A := AutomorphismGroup (G);
> #A;
1024
> A.1;
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1
        G.2 |--> G.2
        G.3 |--> G.1 * G.3
        G.4 |--> G.4
        G.5 |--> G.5
        G.6 |--> G.4 * G.6
> Order (A.1);
4
> a := A.1^2; [a (G.i): i in [1..6]];
[ G.1, G.2, G.3 * G.5, G.4, G.5, G.6 ]
```

`OrderAutomorphismGroupAbelianPGroup(A)`

Order of automorphism group of abelian p-group G where $A = [a_1, a_2, \ldots]$ and $G = C_{a_1} \times C_{a_2} \times \ldots$.

Example H63E25

Subgroups of $C_4 \times C_8 \times C_{64}$.

```
> NumberOfSubgroupsAbelianPGroup ([4, 8, 64]);
[ 7, 35, 91, 139, 171, 171, 139, 91, 35, 7, 1 ]
```

Hence, for example, there are 7 subgroups of order 2 and 139 subgroups of order 2^4.

```
> OrderAutomorphismGroupAbelianPGroup ([4, 8, 64]);
4194304
```

63.12.3 Isomorphism and Standard Presentations

The `pQuotient` command returns a power-conjugate presentation for a given p-group but this presentation depends on the user-supplied description of the group. The Standard Presentation algorithm computes a "canonical" presentation for the p-group, which is independent of the user-supplied description. For a description of this algorithm, see [O'B94].

The canonical or *standard* presentation of a given p-group is the power-conjugate presentation obtained when a description of the group is computed using the default implementation of the p-group generation algorithm.

Hence, two groups in the same isomorphism class have identical standard presentations. Given two p-groups, if their standard presentations are identical, then the groups are isomorphic, otherwise they are not. Hence to decide whether two groups are isomorphic, we can first construct the standard presentation of each using the `StandardPresentation` function and then compare these presentations using the `IsIdenticalPresentation` function.

While it is difficult to state very firm guidelines for the performance of the algorithm, our experience suggests that the difficulty of deciding isomorphism between p-groups is governed by their Frattini rank and is most practical for p-groups of rank at most 5. The *order* of a group is *not* a useful guide to the difficulty of the computation.

`SetVerbose ("Standard", 1)` will provide information on the progress of the algorithm.

`StandardPresentation(G)`

`StandardPresentation(G:` *parameters*`)`

The group G is a p-group presented by an arbitrary pc-presentation. The group H defined by its standard presentation is returned together with a map from G to H.

`StartClass` RngIntElt *Default :* 1

If `StartClass` is k, then use `pQuotient` to construct the class $k-1$ p-quotient of G and standardize the presentation only from class k onwards.

`IsIdenticalPresentation(G, H)`

Returns `true` if G and H have identical presentations, `false` otherwise.

`IsIsomorphic(G, H)`

The function returns `true` if the p-groups G and H are isomorphic, `false` otherwise. It constructs their standard presentations class by class, and checks for equality. If they are isomorphic, it also returns an isomorphism from G to H.

Example H63E26

In the next two examples, we investigate whether particular p-quotients of fp-groups are isomorphic.

```
> F<x, y, t> := FreeGroup(3);
```

```
> G := quo< F | x*y^2*x^-1=y^-2, y*x^2*y^-1=x^-2, x^2=t^2, y^2=(t^-1*x)^2,
>                       t*(x*y)^2=(x*y)^2*t >;
> Q1 := pQuotient(G, 2, 3: Print := 1);
Lower exponent-2 central series for G
Group: G to lower exponent-2 central class 1 has order 2^3
Group: G to lower exponent-2 central class 2 has order 2^7
Group: G to lower exponent-2 central class 3 has order 2^11
> H := quo< F | x*y^2*x^-1=y^-2, y*x^2*y^-1=x^-2, x^2=t^2*(x*y)^2,
>                       y^2=(t^-1*x)^2, t*(x*y)^2=(x*y)^2*t >;
> Q2 := pQuotient(H, 2, 3: Print := 1);
Lower exponent-2 central series for H
Group: H to lower exponent-2 central class 1 has order 2^3
Group: H to lower exponent-2 central class 2 has order 2^7
Group: H to lower exponent-2 central class 3 has order 2^11
```

Now check whether the class 3 2-quotients are isomorphic.

```
> IsIsomorphic(Q1, Q2);
false
```

In the next example, we construct an explicit isomorphism between two 5-groups.

```
> F<a, b> := Group<a, b | a^5, b^5, (a * b * a)^5 = (b, a, b) >;
> G := pQuotient (F, 5, 6 : Print := 1);
Lower exponent-5 central series for F
Group: F to lower exponent-5 central class 1 has order 5^2
Group: F to lower exponent-5 central class 2 has order 5^3
Group: F to lower exponent-5 central class 3 has order 5^4
Group: F to lower exponent-5 central class 4 has order 5^5
Group: F to lower exponent-5 central class 5 has order 5^7
Group: F to lower exponent-5 central class 6 has order 5^8
> G;
GrpPC : G of order 390625 = 5^8
PC-Relations:
G.2^G.1 = G.2 * G.3,
G.3^G.1 = G.3 * G.4,
G.3^G.2 = G.3 * G.6 * G.7^4 * G.8^4,
G.4^G.1 = G.4 * G.5,
G.4^G.2 = G.4 * G.7 * G.8,
G.4^G.3 = G.4 * G.7^4 * G.8,
G.5^G.1 = G.5 * G.6,
G.5^G.2 = G.5 * G.7,
G.5^G.3 = G.5 * G.8^2,
G.6^G.2 = G.6 * G.8,
G.7^G.1 = G.7 * G.8^3
> K<a, b> := Group<a, b | a^5, b^5, (b * a * b)^5 = (b, a, b) >;
> H := pQuotient (K, 5, 6 : Print := 1);
Lower exponent-5 central series for K
Group: K to lower exponent-5 central class 1 has order 5^2
Group: K to lower exponent-5 central class 2 has order 5^3
```

```
Group: K to lower exponent-5 central class 3 has order 5^4
Group: K to lower exponent-5 central class 4 has order 5^5
Group: K to lower exponent-5 central class 5 has order 5^7
Group: K to lower exponent-5 central class 6 has order 5^8
> H;
GrpPC : H of order 390625 = 5^8
PC-Relations:
H.2^H.1 = H.2 * H.3,
H.3^H.1 = H.3 * H.4,
H.3^H.2 = H.3 * H.6^2 * H.7^2 * H.8^2,
H.4^H.1 = H.4 * H.5,
H.4^H.2 = H.4 * H.7,
H.4^H.3 = H.4 * H.7^4 * H.8,
H.5^H.1 = H.5 * H.6,
H.5^H.2 = H.5 * H.7,
H.5^H.3 = H.5 * H.8^2,
H.6^H.2 = H.6 * H.8,
H.7^H.1 = H.7 * H.8^3
> flag, phi := IsIsomorphic (G, H);
> flag;
true
> for g in PCGenerators (G) do print g, "--->", phi (g); end for;
G.1 ---> H.1
G.2 ---> H.2^3 * H.4^3 * H.5^3 * H.6^2 * H.7^4 * H.8^3
G.3 ---> H.3^3 * H.5^3 * H.6^4 * H.8^3
G.4 ---> H.4^3 * H.6^3 * H.7^2 * H.8^3
G.5 ---> H.5^3 * H.8
G.6 ---> H.6^3
G.7 ---> H.7^4
G.8 ---> H.8^4
```

The functions `IsIsomorphic` and `StandardPresentation` are expensive. Here we have a list of groups and we want to find any isomorphisms among the collection. Rather than repeatedly applying `IsIsomorphic`, we first construct and store standard presentations for each group in the sequence, and then quickly compare these using `IsIdenticalPresentation`.

```
> F<a, b> := FreeGroup (2);
> p := 7;
> Q := [];
> for k := 1 to p - 1 do
>       G := quo< F | a^p = (b, a, a), b^p = a^(k*p), (b, a, b)>;
>       H := pQuotient (G, p, 10);
>       Q[k] := StandardPresentation (H);
> end for;
```

Now run over the list of standard presentations and check for equality.

```
> for i in [2..p - 1] do
>    for j in [1.. i - 1] do
>       if IsIdenticalPresentation (Q[i], Q[j]) then
```

```
>          print "Standard Presentations  ", i, " and ", j, " are identical";
>       end if;
>    end for;
> end for;
Standard Presentations   2  and  1  are identical
Standard Presentations   4  and  1  are identical
Standard Presentations   4  and  2  are identical
Standard Presentations   5  and  3  are identical
Standard Presentations   6  and  3  are identical
Standard Presentations   6  and  5  are identical
```

63.13 Generating p-groups

The p-central series of a group G is the descending sequence of subgroups

$$G = P_0(G) \geq \ldots \geq P_{i-1}(G) \geq P_i(G) \geq \ldots \geq$$

where $P_i(G) = [P_{i-1}(G), G]P_{i-1}(G)^p$ for $i \geq 1$.

If $P_c(G) = 1$ and c is the smallest such integer then G has *p-class c*. A group with p-class c is nilpotent and has nilpotency class at most c.

Let G be a finite p-group with Frattini rank d and class c. A group H is a *descendant* of G if H has Frattini rank d and the quotient $H/P_c(H)$ is isomorphic to G. A group is an *immediate descendant* of G if it is a descendant of G and has class $c+1$.

The p-group generation algorithm allows the construction of (immediate) descendants of a p-group. For a description of this algorithm, see [New77, O'B90].

`SetVerbose ("GeneratepGroups", 1)` will provide information on the progress of the algorithm.

`GeneratepGroups (p, d, c : parameters)`

Generate all d-generator p-class at most c p-groups.

`Exponent` RngIntElt *Default :* 0

All groups constructed satisfy the supplied exponent.

`OrderBound` RngIntElt *Default :* 0

Given `OrderBound := n`, all groups constructed have order at most p^n.

`StepSizes` [RngIntElt] *Default :* []

Construct descendants of order $p^{(n+s)}$ of a group of order p^n only for s in `StepSizes`.

`All` BoolElt *Default :* `true`

If `true`, return all groups. Otherwise, return only the capable groups (those which have descendants).

`Descendants(G : parameters)`

`Descendants(G, c : parameters)`

Construct descendants of G having p-class at most c; if c is not supplied, it is assumed to be one larger than the p-class of G. This function supports the same variable arguments as `GeneratepGroups`.

Example H63E27

```
> G := DihedralGroup(GrpPC, 16);
> T := Descendants (G, 8);
> #T;
12
> H := T[5];
> H;
GrpPC : H of order 128 = 2^7
PC-Relations:
    H.1^2 = H.7,
    H.2^2 = H.3 * H.4,
    H.3^2 = H.4 * H.5,
    H.4^2 = H.5 * H.6,
    H.5^2 = H.6 * H.7,
    H.6^2 = H.7,
    H.2^H.1 = H.2 * H.3,
    H.3^H.1 = H.3 * H.4,
    H.4^H.1 = H.4 * H.5,
    H.5^H.1 = H.5 * H.6,
    H.6^H.1 = H.6 * H.7
```

Example H63E28

What is the soluble length of a 2-generator group of exponent 4? We construct the 2-generator 2-groups having exponent 4.

```
> T := GeneratepGroups(2, 2, 10: Exponent := 4);
> "The number of 2-generator exponent 4 groups is ", # T;
The number of 2-generator exponent 4 groups is  26
```

What are their soluble lengths?

```
> for i := 1 to #T do
>       "Group ", i, " has soluble length ", DerivedLength (T[i]);
> end for;
Group  1  has soluble length  1
Group  2  has soluble length  2
Group  3  has soluble length  2
Group  4  has soluble length  1
Group  5  has soluble length  2
Group  6  has soluble length  2
```

```
Group  7  has soluble length  2
Group  8  has soluble length  2
Group  9  has soluble length  2
Group  10  has soluble length  2
Group  11  has soluble length  2
Group  12  has soluble length  2
Group  13  has soluble length  2
Group  14  has soluble length  2
Group  15  has soluble length  2
Group  16  has soluble length  2
Group  17  has soluble length  2
Group  18  has soluble length  2
Group  19  has soluble length  2
Group  20  has soluble length  2
Group  21  has soluble length  3
Group  22  has soluble length  3
Group  23  has soluble length  3
Group  24  has soluble length  3
Group  25  has soluble length  3
Group  26  has soluble length  3
```

Example H63E29

Can we find all 2-generator 3-groups of abundance zero? Such groups have order at most 3^5. First, we define a function which checks the number of conjugacy classes of a group (to determine abundance).

```
> IsGoodGroup := function(G, k)
>
>    ncl := # Classes(G);
>
>    O := FactoredOrder(G);
>    p := O[1][1];
>    m := O[1][2];
>    n := Floor(m / 2);
>    e := m - n * 2;
>    Desired := n * (p^2 - 1) + p^e + k * (p - 1) * (p^2 - 1);
>
>    return (Desired eq ncl);
>
> end function;
```

Then, we generate the potential candidates and check each.

```
> a := GeneratepGroups (3, 2, 4 : OrderBound := 5);
> #a;
42
>
> for i := 1 to #a do
```

```
>        G := a[i];
>        if IsGoodGroup(G, 0) then
>           "Group ", i, " of order ", Order(G), " has abundance 0";
>        end if;
> end for;
Group  1  of order  9  has abundance 0
Group  3  of order  27  has abundance 0
Group  4  of order  27  has abundance 0
Group  11  of order  81  has abundance 0
Group  12  of order  81  has abundance 0
Group  13  of order  81  has abundance 0
Group  14  of order  81  has abundance 0
Group  40  of order  243  has abundance 0
Group  41  of order  243  has abundance 0
Group  42  of order  243  has abundance 0
```

`ClassTwo(p, d :` *parameters*`)`

`ClassTwo(p, d, Step :` *parameters*`)`

`ClassTwo(p, d, s :` *parameters*`)`

Count the d-generator p-groups of p-class 2. If s or *Step* is supplied, then count only those of order $p^{(d+s)}$ or $p^{(d+m)}$ for $m \in Step$. In the first two invocations, the sequence returns a sequence of length $\binom{d}{2}$, whose m-th entry is the number of groups of $p^{(d+m)}$. (Some additional entries may be deduced on the basis of duality.) The last invocation returns the number of groups of $p^{(d+s)}$. For details of the algorithm used see [EO99].

`Exponent` RngIntElt *Default :* 0

If `Exponent` is `true`, count those groups which have exponent p. The directive `SetVerbose ("ClassTwo", 1)` will provide information on the progress of the algorithm.

Example H63E30

Count the number of 3-generator p-class 2 5-groups.

```
> ClassTwo(5, 3);
[ 4, 19, 42, 19, 4, 1 ]
```

For example, the number of 3-generator 5-groups of order 5^6 and p-class 2 is precisely 42.
Count the number of 4-generator p-class 2 5-groups of order 5^7.

```
> ClassTwo(5, 4, 3);
6598
```

63.14 Representation Theory

Chapter 91 on characters describes many functions for computing with partial character tables or individual characters.

`CharacterDegrees(G)`

`CharacterDegrees(G, z, p)`

Given a finite pc-group G, return the sequence $[\langle d_1, c_1\rangle, \langle d_2, c_2\rangle, \ldots]$, where c_i is the number of irreducible characters of G having degree d_i. For details of the algorithm see Conlon [Con90b].

The second form requires z to be a central element of G and p to be a prime or zero. The sequence returned enumerates the number of absolutely irreducible characters of G in characteristic p, lying over some faithful linear character of $\langle z\rangle$.

`CharacterDegrees(G)`

Given a finite p-group G, return the sequence $[\langle d_1, c_1\rangle, \langle d_2, c_2\rangle, \ldots]$, where c_i is the number of irreducible characters of G having degree d_i. For details of the algorithm see [Sla86].

`CharacterDegreesPGroup(G)`

Given a finite p-group G, return the sequence $[C_0, C_1, \ldots]$, where C_i is the number of irreducible characters of G having degree p^i. For details of the algorithm see [Sla86].

`CharacterTable(G:` *parameters*`)`

Construct the table of ordinary irreducible characters for the group G.

`Al` MonStgElt *Default :* "*Default*"

This parameter controls the algorithm used. The string `"DS"` forces use of the Dixon-Schneider algorithm. The string `"IR"` forces the use of Unger's induction/reduction algorithm [Ung06]. The `"Default"` algorithm is to use Dixon-Schneider for groups of order ≤ 5000 and Unger's algorithm for larger groups. This may change in future.

`DSSizeLimit` RngIntElt *Default :* 10^4

When the default algorithm is selected, a positive value n for `DSSizeLimit` means that before using Unger's algorithm, the full character space is split by some passes of Dixon-Schneider, restricted to using class matrices corresponding to conjugacy classes with size at most n.

`CharacterTableConlon(G)`

Given a finite p-group G, return the character table of G. The algorithm is due to Conlon, as described in [Con90].

`GModule(G, M)`

The G-module for the action of G on the vector space defined by the matrix ring M.

`GModule(G, A)`

A KG-module M corresponding to the action of the group G on the elementary abelian subgroup A of G is constructed. The map from A to the vector space underlying M is also returned.

`GModule(G, A, B)`

A KG-module M corresponding to the action of the group G on the elementary abelian section A/B of G is constructed. The map from A to the vector space underlying M is also returned.

`AbsolutelyIrreducibleRepresentationsSchur(G, k:` *parameters*`)`

`AbsolutelyIrreducibleModulesSchur(G, k:` *parameters*`)`

Compute the absolutely irreducible representations of the group G over appropriate extensions or sub-fields of the given field k. The representations returned are inequivalent and consist of all distinct representations, subject to the conditions imposed. The field k may be a finite field, the rationals or a cyclotomic field. In the case when k is a finite field, the Glasby-Howlett algorithm is used to determine the minimal field over which a representation may be realised. If k has characteristic 0, the field over which a representation is realised may not be minimal.

The representations are found using Schur's method of climbing the composition series for G defined by the pc-presentation. If the argument i is given then the algorithm will calculate only representations of the ith subgroup of the composition series.

The "Representations" function returns a list of homomorphisms $\rho : G \to GL(n, K)$, where K is a field compatible with k. The "Modules" version returns an equivalent list of G-modules.

`Process` BoolElt *Default :* `true`

If the parameter `Process` is set true then the list is a list of pairs comprising an integer and a representation. This list or any sublist of it is a suitable value for the argument L in the last versions of the function, and in this case only the representations in L will be extended up the series. This allows the user to inspect the representations produced along the way and cull any that are uninteresting.

`GaloisAction` MonStgElt *Default :* "Yes"

Possible values are `"Yes"`, `"No"` and `"Relative"` The default is `"Yes"` for intermediate levels and `"No"` for the whole group. The value `"Yes"` means that it only lists one representation from each orbit of the action of the absolute Galois group $Gal(K/\text{primefield}(K))$. Setting this parameter to `"No"` turns this reduction off (thus listing all inequivalent representations), while setting it to `"Relative"` uses the group $Gal(K/k)$.

MaxDimension RngIntElt *Default :*

Restrict the representations to those of dimension $\leq$ MaxDimension. Default is no restriction.

ExactDimension SetEnum *Default :*

If ExactDimension is assigned a set S of positive integers, attention is restricted to representations having dimensions lying in the set S. The default is equivalent to taking the set of all positive integers.

If both MaxDimension and ExactDimension are assigned values, then representations having dimensions that are either bounded by MaxDimension or contained in ExactDimension are produced.

IrreducibleRepresentationsSchur(G, k: *parameters*)

IrreducibleModulesSchur(G, k: *parameters*)

Compute irreducible representations of G over the given field k. All arguments and parameters are as for the absolutely irreducible case.

The computation proceeds by first computing the absolutely irreducible representations subject to the given parameters, then rewriting over the field k, with a consequent change of dimension of the representation.

Example H63E31

We compute representations of the dihedral group of order 20.

```
> G := DihedralGroup(GrpPC, 10);
> FactoredOrder(G);
[ <2, 2>, <5, 1> ]
```

First some modular representations with characteristic 2.

```
> r := IrreducibleModulesSchur(G, GF(2));
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2)
*]
> r := AbsolutelyIrreducibleModulesSchur(G, GF(2));
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(2^2),
    GModule of dimension 2 over GF(2^2)
*]
> r := AbsolutelyIrreducibleModulesSchur(G, GF(2) : GaloisAction:="Yes");
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(2^2)
```

```
*]
```

The irreducible representation of dimension 4 is not absolutely irreducible, as over $GF(4)$ it splits into two Galois-equivalent representations.
Getting irreducible representations over the complex field presents no problem, despite not being able to use the complex field as an argument to the function call. We could specify the field to be the cyclotomic field with degree equal to Exponent(G), but it is preferable to ask for absolutely irreducible representations over the rationals.

```
> r := AbsolutelyIrreducibleRepresentationsSchur(G, Rationals());
> r;
[*
    Mapping from: GrpPC: G to GL(1, RationalField()),
    Mapping from: GrpPC: G to GL(1, RationalField()),
    Mapping from: GrpPC: G to GL(1, RationalField()),
    Mapping from: GrpPC: G to GL(1, RationalField()),
    Mapping from: GrpPC: G to GL(2, CyclotomicField(5)),
    Mapping from: GrpPC: G to GL(2, CyclotomicField(5)),
    Mapping from: GrpPC: G to GL(2, CyclotomicField(5)),
    Mapping from: GrpPC: G to GL(2, CyclotomicField(5))
*]
> r[6](G.2);
[zeta_5^3        0]
[       0 zeta_5^2]
```

63.15 Central Extensions

We now describe functions to construct $H^2(G, U)$ for a finite soluble group G and finite abelian group U (a trivial G-module). We also present functions to construct central extensions of U by G.

Denote by $Z^2(G, U)$ the abelian group of all cocycles from G to U, under pointwise multiplication. The values $\psi(g, h)$ of $\psi \in Z^2(G, U)$ may be represented as a "cocyclic matrix" with entries in U.

If $\phi : G \to U$ is a set map with $\phi(1_G) = 1_U$, then there is a coboundary $\partial\phi \in Z^2(G, U)$ defined by $\partial\phi(g, h) = \phi(g)\phi(h)\phi(gh)^{-1}$. The group of all coboundaries from G to U is denoted $B^2(G, U)$, and we have $H^2(G, U) = Z^2(G, U)/B^2(G, U)$. Then $H^2(G, U) = I \times T$, where I is the (faithful) image of $\mathrm{Ext}(G/G', U) \leq H^2(G/G', U)$ under inflation, and T is the (faithful) image of $\mathrm{Hom}(H_2(G), U)$ under a certain transgression homomorphism. Here we provide functions which construct representatives for the elements in a generating set for each of these two factors.

For details of the theory and the algorithm used, see [FO00].

`SetVerbose ("Cocycle", 1)` will provide additional information on the calculations in the functions.

`ExtGenerators(G, U)`

Given a soluble group G and an abelian group U (both defined by pc-presentations) the function returns a sequence of tuples describing generators for $\mathrm{Ext}(G/G', U)$ as cocyclic matrices; the first entry in each tuple is a representative of a generator, the second is the order of the coset of the representative in $H^2(G, U)$.

`HomGenerators(G, U)`

Given a soluble group G and an abelian group U (both defined by pc-presentations) the function returns a sequence of tuples describing generators for $\mathrm{Hom}(H_2(G), U)$ as cocyclic matrices; the first entry in each tuple is a representative of a generator, the second is the order of the coset of the representative in $H^2(G, U)$.

`ElementSequence(G)`

For a soluble group G, the function returns an indexed set of elements of G listed in the order used by `ExtGenerators` and `HomGenerators`.

`RepresentativeCocycles(G, U, Ext, Hom)`

Let G be a soluble group G and U be an abelian group both defined by pc-presentations. Let Ext and Hom be the values returned by calling `ExtGenerators` and `HomGenerators` respectively. The function `RepresentativeCocycles` returns a complete and irredundant set of representatives for the elements of $H^2(G, U)$ as cocyclic matrices.

`CentralExtension(G, U, A)`

Let G be a soluble group G and U be an abelian group, both defined by pc-presentations. Further, let A be a cocyclic matrix (as determined by the function `RepresentativeCocycles`). Then, this function returns the central extension of U by G determined by the cocyclic matrix A.

`CentralExtensions(G, U, Q)`

If G is a soluble group G and U is an abelian group, both defined by pc-presentations, and Q is a sequence of cocyclic matrices (as determined by the function `RepresentativeCocycles`), this function returns the corresponding sequence of central extension of U by G determined by the sequence of cocyclic matrices A. Note that the central extensions thereby constructed need not be mutually non-isomorphic.

`CentralExtensionProcess(G, U)`

Given a soluble group G and an abelian group U (both defined by pc-presentations) the function creates a process P for central extensions of U by G. Note that the list of central extensions constructed by this process will contain all isomorphism types but the extensions need not be mutually non-isomorphic.

`NextExtension(~P)`

Given a central extension process P, construct the next central extension determined by P.

`IsEmpty(P)`

Return `true` if all central extensions determined by the process P have been constructed; otherwise return `false`.

Example H63E32

We compute the abelian invariants of $H^2(D_4, C_2)$.

```
> G := DihedralGroup(GrpPC, 4);
> U := AbelianGroup(GrpPC, [2]);
>
> Ext := ExtGenerators(G, U);
> Ext[1];
    <[Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) Id(U) Id(U) Id(U) U.1 U.1 U.1 U.1]
    [Id(U) Id(U) Id(U) Id(U) U.1 U.1 U.1 U.1]
    [Id(U) Id(U) Id(U) Id(U) U.1 U.1 U.1 U.1]
    [Id(U) Id(U) Id(U) Id(U) U.1 U.1 U.1 U.1], 2>,
>
> Hom := HomGenerators(G, U);
> Hom;
[
    <[Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) U.1 U.1 Id(U) Id(U) U.1 U.1 Id(U)]
    [Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) U.1 U.1 Id(U) Id(U) U.1 U.1 Id(U)]
    [Id(U) Id(U) Id(U) U.1 Id(U) U.1 U.1 U.1]
    [Id(U) U.1 Id(U) Id(U) U.1 U.1 Id(U) U.1]
    [Id(U) Id(U) Id(U) U.1 Id(U) U.1 U.1 U.1]
    [Id(U) U.1 Id(U) Id(U) U.1 U.1 Id(U) U.1], 2>
]
>
> AbelianInvariants(Ext, Hom);
[ 2, 2, 2 ]
```

We now compute the central extension of U by G determined by a single cocyclic matrix.

```
> A := RepresentativeCocycles(G, U, Ext, Hom);
> E := CentralExtension(G, U, A[2]);
> E;
GrpPC : E of order 16 = 2^4
PC-Relations:
```

```
    E.1^2 = E.4,
    E.2^2 = E.3 * E.4,
    E.2^E.1 = E.2 * E.3
```

Alternatively we can build all central extensions of U by G.

```
> E := CentralExtensions(G, U, A);
> "Number of extensions is ", #E;
Number of extensions is  8
```

Next, we provide an example of using the central extension process. Firstly, we create the groups and initialize the process.

```
> G := SmallGroup(12, 5);
> U := AbelianGroup(GrpPC, [2, 3]);
> P := CentralExtensionProcess(G, U);
```

Now we run over the central extensions and count conjugacy classes.

```
> C := [];
> while IsEmpty(P) eq false do
>    NextExtension(~P, ~E);
>    Append(~C, #Classes (E));
> end while;
> "# conjugacy classes is ", C;
# conjugacy classes is  [ 45, 72, 45, 72, 45, 72, 45, 72, 45, 72, 45,
72, 45, 72, 45, 72, 45, 72, 45, 72, 45, 72, 45, 72 ]
```

63.16 Transfer Between Group Categories

63.16.1 Transfer to GrpPC

The `PolycyclicGroup`-constructor allows complete flexibility in defining a pc-group. However, it is often more convenient to have MAGMA compute a pc-presentation based on some other description of the group. The `PCGroup` function will produce a pc-presentation for a finite group in various categories such as `GrpPerm` and `GrpMat`. Converting from a `GrpFP` group is trickier, since the original group need not be finite. There are two functions provided to produce pc-presentations for certain quotients of finitely-presented groups. The `pQuotient` function constructs a pc-presentation for the largest p-group quotient having specified lower exponent-p class. Similarly, `SolubleQuotient` will compute the largest soluble quotient subject to certain restrictions. Each of these functions also provides a homomorphism (isomorphism in the case of `PCGroup`) from the original group to the new pc-group. More information on each of the two quotient functions can be found in Chapter 71.

`PCGroup(G)`

A `GrpPC` representation of the group G and the isomorphism.

`pQuotient( F, p, c : parameters)`

`Workspace`	RngIntElt	*Default :* 1000000
`Metabelian`	BoolElt	*Default :* `false`
`Exponent`	RngIntElt	*Default :* 0
`Print`	RngIntElt	*Default :* 0

Given a finitely presented group F, a prime p, and a positive integer c, this function constructs a consistent power-conjugate presentation for the largest p-quotient H of F having lower exponent-p class at most c). If c is given as zero, then the limit 127 is placed on the class. The function returns both the p-quotient H defined by a pc-presentation and the homomorphism from F to H.

`SolubleQuotient(G)`

`SolvableQuotient(G)`

A `GrpPC` representation P of the largest solvable quotient of G and the homomorphism $\phi : G \to P$.

Example H63E33

We use `PCGroup` to produce a pc-presentation for a matrix group.

```
> GL := GeneralLinearGroup(4,GF(3));
> S3 := Sylow(GL,3);
> P := PCGroup(S3);
> P;
GrpPC : P of order 729 = 3^6
PC-Relations:
    P.2^P.1 = P.2 * P.4^2,
    P.3^P.1 = P.3 * P.5^2,
    P.3^P.2 = P.3 * P.6^2,
    P.5^P.2 = P.4 * P.5,
    P.6^P.1 = P.4 * P.6
```

63.16.2 Transfer from GrpPC

Given a pc-group, it is straight-forward to convert it to a `GrpFP` or `GrpGPC` representation by using the appropriate transfer function. If one wishes to have a permutation representation of the group, this requires more cleverness. The `CosetAction` function can be used to compute the permutation representation of a group on a subgroup. If the subgroup is chosen to have trivial core, then the permutation group obtained will be isomorphic to the original group.

`AbelianGroup(G)`

Given an abelian pc-group G, return a `GrpAb` group H isomorphic to G and an isomorphism $\phi : G \to H$.

FPGroup(G)

A GrpFP representation F of G and the isomorphism from G to F.

GPCGroup(G)

A GrpGPC representation F of G and the isomorphism from G to F.

Example H63E34

Take one of the groups of order $2^6 * 3^2$.

```
> G := SmallGroup(576, 4123);
> G;
GrpPC : G of order 576 = 2^6 * 3^2
PC-Relations:
    G.1^2 = Id(G),
    G.2^2 = Id(G),
    G.3^2 = G.5,
    G.4^3 = Id(G),
    G.5^2 = G.7,
    G.6^2 = G.7,
    G.7^2 = Id(G),
    G.8^3 = Id(G),
    G.2^G.1 = G.2 * G.6,
    G.6^G.1 = G.6 * G.7,
    G.6^G.2 = G.6 * G.7,
    G.8^G.1 = G.8^2
```

Since G is small, we can search for a minimum degree permutation presentation by brute force. First we build a set containing all the subgroups.

```
> SL := Subgroups(G);
> T := {X`subgroup: X in SL};
> #T;
243
```

Then, we select those subgroups with trivial core, and find one with the smallest index.

```
> TrivCore := {H:H in T| #Core(G,H) eq 1};
> mdeg := Min({Index(G,H):H in TrivCore});
> Good := {H: H in TrivCore| Index(G,H) eq mdeg};
> #Good;
3
> H := Rep(Good);
```

We then use CosetAction to construct the permutation representation on the cosets of H.

```
> f,P,K := CosetAction(G,H);
> #K;
1
> IsPrimitive(P);
false
```

63.17 More About Presentations

Each pc-group can have up to three pc-presentations associated with it. If the user specifies a consistent presentation in the `PolycyclicGroup`-constructor, then this presentation (the "user" presentation) will be used for all printing and interpretation of element input. If the specified presentation is inconsistent, a runtime error is generated.

Internally, MAGMA uses a "conditioned" presentation for computation. The composition series associated with this presentation is guaranteed to refine a normal series with elementary abelian factors. If G is a p-group, then the composition series is guaranteed to be a central series and the first d pc-generators are a minimal set of generators for the group. Hence, their images generate the Frattini factor group. If the user presentation satisfies these conditions, then it is used as the conditioned presentation. Otherwise, a separate presentation is computed automatically.

Several algorithms rely on a "special" presentation for the group. This presentation exhibits Hall π-subgroups and a characteristic series with elementary abelian factors. When needed, such a presentation is computed and elements are automatically translated between presentations.

The "compact" presentation is not a presentation used in computation. Rather it provides an efficient means to input and output large pc-groups. This is especially useful for stored collections of groups (libraries or databases).

63.17.1 Conditioned Presentations

MAGMA will compute a pc-presentation which will be used for internal computation, but the user's presentation will be used for all input and output. The recommended way to access the conditioned internal presentation is via the intrinsic `ConditionedGroup`.

63.17.1.1 Structure Operations

`ConditionedGroup(G)`

The internally used, conditioned presentation of the pc-group G. The returned group is recorded as a subgroup of G in the relationship tables, so coercion can be used to move between presentations.

`IsConditioned(G)`

Reutrns `true` if G uses the user presentation as the internal presentation, `false` otherwise.

63.17.1.2 Element Operations

`LeadingTerm(x)`

Given an element x of a pc-group G with n pc-generators and a conditioned presentation, where x is of the form $a_1^{\alpha_1} \dots a_n^{\alpha_n}$, return $a_i^{\alpha_i}$ for the smallest i such that $\alpha_i > 0$. If x is the identity of G, then the identity is returned.

`LeadingGenerator(x)`

Given an element x of a pc-group G with n pc-generators and a conditioned presentation, where x is of the form $a_1^{\alpha_1} \dots a_n^{\alpha_n}$, return a_i for the smallest i such that $\alpha_i > 0$. If x is the identity of G, then the identity is returned.

`LeadingExponent(x)`

Given an element x of a pc-group G with n pc-generators and a conditioned presentation, where x is of the form $a_1^{\alpha_1} \dots a_n^{\alpha_n}$, return α_i for the smallest i such that $\alpha_i > 0$. If x is the identity of G, then 0 is returned.

`Depth(x)`

Given an element x of a pc-group G with n pc-generators and a conditioned presentation, where x is of the form $a_1^{\alpha_1} \dots a_n^{\alpha_n}$, return the smallest i such that $\alpha_i > 0$. If x is the identity of G, then 0 is returned.

`PCClass(x)`

`WeightClass(x)`

The weight class of the element x. The `WeightClass` of an arbitrary element of a pc-group G is defined to be k if $x \in G_{\delta_{k-1}}$ and $x \notin G_{\delta_k}$. If x is the identity of G, then `WeightClass` returns $n + 1$.

63.17.2 Special Presentations

A special presentation is one which has several properties described by C. R. Leedham-Green:

(1) The composition series defined by the pc-generators refines the LG-series. The LG-series is a characteristic series which refines the nilpotent series. Within each nilpotent section, it refines the series of successive Frattini factors. Factors of successive terms in the LG-series are elementary abelian p-groups, with p increasing through each Frattini factor.

(2) The presentation exhibits a Sylow system. By this we mean that if π is a set of primes, then the pc-generators whose corresponding prime lies in π will generate a Hall π-subgroup.

(3) The presentation exhibits "head splittings". These are certain complements in factors of the group as follows: If N is a term of the nilpotent series of G, M the next term (so N/M is a maximal nilpotent factor of N), and F/M is the Frattini subgroup of N/M, then it is possible to show that N/F has a complement in G/F. We say the presentation exhibits this complement (or "splitting") if the pc-generators of G which are not in N generate a complement for N mod F.

Several algorithms rely on having a special presentation for the given group. In these cases, MAGMA will automatically compute a special presentation. However, if the user wishes to have a special presentation as the user presentation for a group, the function

SpecialPresentation can be used. This is typically used when implementing new algorithms which rely on the properties of a special presentation. The other functions allow one to identify specific characteristics of a special presentation. They are not defined for arbitrary presentations.

`SpecialPresentation(G)`

Returns a new group H which is defined by a special presentation. H is in fact a subgroup of G (equal to G) and so one can use the coercion operator (!) to translate elements between the two presentations. Furthermore, any subgroup of H is automatically a subgroup of G. For instance, if one computed the center Z of H (using some algorithm relying on the special presentation), Z would be a subgroup of G, and would be the center of G.

`SpecialWeights(G)`

A sequence of triples of integers is returned, with one triple corresponding to each pc-generator. The first integer in a triple gives the number of the nilpotent section containing the generator, the second gives the number of the square-free exponent abelian section of that nilpotent section containing it, and the third gives the number of the elementary abelian p-group layer that contains the generator. The prime for the generator is not included in the triple (see `PCPrimes`).

`NilpotentLength(G)`

The number of nilpotent factors in the nilpotent series.

`NilpotentBoundary(G,i)`

The subscript of the last generator in the ith nilpotent section, where i lies between 1 and `NilpotentLength(G)`.

`MinorLength(G,i)`

The number of minor sections (Frattini factors) in the ith nilpotent section of G.

`MinorBoundary(G,i,j)`

The subscript of the last generator in the jth minor section of the ith nilpotent section, where j lies between 1 and `MinorLength(G,i)`.

`LayerLength(G,i,j)`

The number of elementary abelian p-group layers in the jth minor section of the ith nilpotent section of G.

`LayerBoundary(G,i,j,k)`

The subscript of the last generator in the kth elementary abelian p-group layer of the jth minor section of the ith nilpotent section, where k lies between 1 and `LayerLength(G,i,j)`.

Example H63E35

We show how user presentations and special presentations can differ. If we define a wreath product using `PolycyclicGroup`, the given presentation becomes the user presentation, but this is not a special presentation for the group.

```
> T := PolycyclicGroup<a,b,c,d|a^3,b^3,c^3,d^3,
>                          b^a=c, c^a=d, d^a=b>;
> T;
GrpPC : T of order 81 = 3^4
PC-Relations:
    T.2^T.1 = T.3,
    T.3^T.1 = T.4,
    T.4^T.1 = T.2
> S := SpecialPresentation(T);
> S;
GrpPC : S of order 81 = 3^4
PC-Relations:
    S.2^S.1 = S.2 * S.3^2 * S.4,
    S.3^S.1 = S.3 * S.4^2
```

Here we build another wreath product and construct a special presentation.

```
> C6 := CyclicGroup(GrpPC,6);
> C2 := CyclicGroup(GrpPC,2);
> G := WreathProduct(C2,C6);
> G;
GrpPC : G of order 384 = 2^7 * 3
PC-Relations:
    G.1^2 = G.2,
    G.2^3 = Id(G),
    G.3^2 = Id(G),
    G.4^2 = Id(G),
    G.5^2 = Id(G),
    G.6^2 = Id(G),
    G.7^2 = Id(G),
    G.8^2 = Id(G),
    G.3^G.1 = G.8,
    G.3^G.2 = G.5,
    G.4^G.1 = G.6,
    G.4^G.2 = G.3,
    G.5^G.1 = G.7,
    G.5^G.2 = G.4,
    G.6^G.1 = G.3,
    G.6^G.2 = G.8,
    G.7^G.1 = G.4,
    G.7^G.2 = G.6,
    G.8^G.1 = G.5,
    G.8^G.2 = G.7
> H := SpecialPresentation(G);
```

```
> H;
GrpPC : H of order 384 = 2^7 * 3
PC-Relations:
    H.1^2 = Id(H),
    H.2^2 = Id(H),
    H.3^3 = Id(H),
    H.4^2 = Id(H),
    H.5^2 = Id(H),
    H.6^2 = Id(H),
    H.7^2 = Id(H),
    H.8^2 = Id(H),
    H.2^H.1 = H.2 * H.4,
    H.5^H.3 = H.6,
    H.6^H.3 = H.5 * H.6,
    H.7^H.1 = H.6 * H.7,
    H.7^H.3 = H.8,
    H.8^H.1 = H.5 * H.6 * H.8,
    H.8^H.3 = H.7 * H.8
```

We can coerce between the presentations.

```
> G!(H.2), H!(G.2);
G.6 * G.7 * G.8 H.3
```

Look at some specific features of the presentation.

```
> SpecialWeights(H);
[ <1, 1, 1>, <1, 1, 1>, <1, 1, 2>, <1, 2, 1>, <2, 1, 1>, <2, 1, 1>, <2, 1, 1>,
<2, 1, 1> ]
> MinorLength(H,1);
2
> MinorBoundary(H,1,1);
3
```

63.17.3 CompactPresentation

When the MAGMA parser reads in large group presentations of the form

```
S4 := PolycyclicGroup< a, b, c, d | a^2 = 1, b^3 = 1, c^2 = 1,
       d^2 = 1, b^a = b^2, c^a = c * d, c^b = c * d, d^b = c >;
```

a large amount of memory and time is used to build all of the expressions involved in the statement. This time is most noticeable when loading in large libraries of MAGMA code containing many large presentations. The following intrinsics provide a way to avoid this overhead.

`CompactPresentation(G)`

Given a pc-group G, return a sequence of integers that contains the information needed to define the group's presentation.

PCGroup(Q : *parameters*)

Check	BoolElt	*Default :* false
ExponentLimit	RngIntElt	*Default :* 20

Return a group G in category GrpPC, whose presentation is provided by the integer sequence Q. Constructing the group from the integer sequence has very low overhead in the parser. The time taken to construct the group is less when the presentation is conditioned.

The parameter Check indicates whether or not the presentation is checked for consistency. Leaving the Check parameter set to false speeds the construction of the group, but will be disastrous if the sequence Q does not represent a consistent pc-presentation.

Parameter ExponentLimit determines the amount of space that will be used by the group to speed calculations. Given ExponentLimit := e, the group will store the products $a^i * b^j$ where a and b are generators and i and j are in the range 1 to e.

Example H63E36

If the user wants to store the definition of a group in a library, the following may be done.

```
> S4 := PolycyclicGroup< a, b, c, d | a^2 = 1, b^3 = 1, c^2 = 1, d^2 = 1,
>  b^a = b^2, c^a = c * d, c^b = c * d, d^b = c >;
> Q := CompactPresentation( S4 );
> Q;
[ 4, -2, -3, -2, 2, 33, 218, 114, 55 ]
```

The library code would then be

```
> Make:=func< | PCGroup(\[4, 2, 3, 2, 2, 33, 218, 114, 55] : Check := false) >;
```

Note the use of a literal sequence here — see Chapter 10.

63.18 Optimizing Magma Code

63.18.1 PowerGroup

If the user is working with enumerated sets of pc-groups that are all subgroups of a common over-group G, then the following optimization is strongly recommended. Define the set to have the universe PowerGroup(G). For any subgroup H of G, we can find a canonical form for the generators of H. This allows us to have a very good hashing function for the subgroups.

Example H63E37

The following example illustrates the optimization.

```
> G := ExtraSpecialGroup( GrpPC, 3, 3 );
> P := PowerGroup(G);
> time s1 := {  P | sub< G | Random(G), Random(G) > : x in { 1..500}  };
Time: 1.140
> time s2 := { Parent(G) | sub< G | Random(G), Random(G) > : x in { 1..500}  };
Time: 9.769
```

63.19 Bibliography

[**CH00**] John Cossey and Trevor Hawkes. On the largest conjugacy class size in a finite group. *Rend. Sem. Mat. Univ. Padova*, 103:171–179, 2000.

[**CNW90**] F. Celler, J. Neubüser, and C.R.B. Wright. Some remarks on the computation of complements and normalizers in soluble groups. *Acta Appl. Math.*, 21:57–76, 1990.

[**Con90a**] S. B. Conlon. Calculating characters of p-groups. *J. Symbolic Comp.*, 9:535–550, 1990.

[**Con90b**] S. B. Conlon. Computing modular and projective character degrees of soluble groups. *J. Symbolic Comp.*, 9:551–570, 1990.

[**ELGO02**] Bettina Eick, C.R. Leedham-Green, and E.A. O'Brien. Constructing automorphism groups of a p-groups. *Comm. Algebra*, 30:2271–2295, 2002.

[**EO99**] Bettina Eick and E.A. O'Brien. Enumerating p-groups. *J. Austral. Math. Soc.*, 67:191–205, 1999.

[**FO00**] D.L. Flannery and E.A. O'Brien. Computing 2-cocycles for central extensions and relative difference sets. *Comm. Algebra*, 28:1935–1955, 2000.

[**GS90**] S.P. Glasby and Michael C. Slattery. Computing intersections and normalizers in soluble groups. *J. Symbolic Comp.*, 9:637–651, 1990. Computational group theory, Part 1.

[**How12**] David J. A. Howden. *Computing automorphism groups and isomorphism testing in finite groups*. PhD thesis, University of Warwick, 2012.

[**Hul99**] Alexander Hulpke. Computing subgroups invariant under a set of automorphisms. *J. Symbolic Comp.*, 27:415–427, 1999.

[**MN89**] M. Mecky and J. Neubüser. Some remarks on the computation of conjugacy classes of soluble groups. *Bull. Austral, Math. Soc.*, 40(2):281–292, 1989.

[**New77**] M.F. Newman. Determination of groups of prime-power order. In *Group Theory (Canberra, 1975)*, volume 573 of *Lecture Notes in Mathematics*, pages 73–84. Springer-Verlag, Berlin-Heidelberg-New York, 1977.

[**O'B90**] E.A. O'Brien. The p-group generation algorithm. *J. Symbolic Comput.*, 9:677–698, 1990.

[**O'B94**] E.A. O'Brien. Isomorphism testing for p-groups. *J. Symbolic Comp.*, 17:133–147, 1994.

[**Sla86**] Michael C. Slattery. Computing character degrees in p-groups. *J. Symbolic Comp.*, 2:51–58, 1986.

[**Sla01**] Michael C. Slattery. Computing double cosets in soluble groups. *J. Symbolic Comp.*, 31:179–192, 2001. Computational algebra and number theory (Milwaukee, WI, 1996).

[**Smi94**] Michael J. Smith. *Computing automorphisms of finite soluble groups.* PhD thesis, Australian National University, 1994.

[**Ung06**] W.R. Unger. Computing the character table of a finite group. *J. Symbolic Comp.*, 41(8):847–862, 2006.

64 BLACK-BOX GROUPS

Chapter 64
BLACK-BOX GROUPS

64.1 Introduction

This Chapter describes the category of black-box groups (BB-groups). The name in MAGMA for the category of BB-groups is `GrpBB`.

64.2 Construction of an SLP-Group and its Elements

64.2.1 Structure Constructors

Magma's black-box groups are built on Magma's other group types. The basic constructor takes a group and returns a corresponding black-box group. The element set of the black-box group is essentially the same as the element set of the original group, and the group operations are inherited from the original group.

`NaturalBlackBoxGroup(H)`

Construct the natural black-box group from the concrete group H.

64.2.2 Construction of an Element

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity element for the BB-group G.

64.3 Arithmetic with Elements

`u * v`

Construct the product of elements u and v of the BB-group G.

`u ^ m`

Given an integer m and u, an element of BB-group G, return the element of G corresponding to the m-th power of u.

`u ^ v`

Given u and v, elements of BB-group G, return the element of G corresponding to the conjugate of u by v, i.e. $v^{-1} * u * v$.

`(u, v)`

Commutator of the elements u and v, i.e. the element $u^{-1} * v^{-1} * u * v$. Here u and v must belong to the same BB-group G.

64.3.1 Accessing the Defining Generators

The functions described here provide access to basic information stored for a BB-group G.

`G . i`

The i-th generator for G.

`Generators(G)`

A set containing the generators for G.

`NumberOfGenerators(G)`
`Ngens(G)`

The number of generators for B.

64.4 Operations on Elements

64.4.1 Equality and Comparison

`u eq v`

Returns `true` if and only if the underlying concrete group elements for u and v are equal.

`u ne v`

Returns `true` if and only if the underlying concrete group elements for u and v are not equal.

64.4.2 Attributes of Elements

`Parent(u)`

The parent group G of the element u.

`UnderlyingElement(u)`

The concrete group element corresponding to the BB-group element u.

`Order(u)`

The order of the underlying concrete group element of u.

Example H64E1

The following function takes a black box group isomorphic to M_{24} and finds standard generators. It is taken from the ATLAS of Finite Group Representations page on M_{24}.

```
> m24_standard := function(B)
> repeat a := PseudoRandom(B); until Order(a) eq 10;
> a := a ^ 5;
> repeat b := PseudoRandom(B); until Order(b) eq 15;
> b := b ^ 5;
> repeat b := b ^ PseudoRandom(B); ab := a*b;
> until Order(ab) eq 23;
> x := ab*(ab^2*b)^2*ab*b;
> if Order(x) eq 5 then b := b^-1; end if;
> return a,b;
> end function;
```

We take a group which must be M_{24} and find these generators.

```
> G := PermutationGroup<24 |
> [ 20, 4, 10, 3, 15, 9, 7, 1, 11, 22, 21, 19, 8, 2, 24, 5,
> 12, 18, 13, 16, 14, 23, 6, 17 ],
> [ 12, 18, 3, 2, 7, 11, 5, 21, 19, 22, 23, 1, 14, 17, 10,
> 8, 4, 13, 24, 20, 9, 15, 6, 16 ]>;
> #G;
244823040
> Transitivity(G);
5
> B := NaturalBlackBoxGroup(G);
> a,b := m24_standard(B); a,b;
GrpBBElt (1, 16)(2, 22)(3, 14)(4, 15)(5, 11)(6, 24)(7,
10)(8, 18)(9, 19)(12, 17)(13, 20)(21, 23)
GrpBBElt (1, 14, 17)(2, 18, 13)(5, 16, 20)(7, 22, 9)(8, 24,
15)(19, 23, 21)
```

The printing of the GrpBBElts shows the underlying concrete group elements. These may be extracted using the `UnderlyingElement` intrinsic for use within G.

64.5 Set-Theoretic Operations

64.5.1 Membership and Equality

`g in G`

Return true if and only if G is the parent group of g or the parent group of g is a subgroup of G.

64.5.2 Set Operations

`PseudoRandom(G)`

Return a pseudo-random element of the BB-group G. The method used is product-replacement with accumulator.

`Rep(G)`

A representative element of G.

64.5.3 Coercions Between Related Groups

`G ! g`

Given an element g belonging to a subgroup of the BB-group G, rewrite g as an element of G.

65 ALMOST SIMPLE GROUPS

Chapter 65
ALMOST SIMPLE GROUPS

65.1 Introduction

65.1.1 Overview

This chapter describes a set of tools for working with finite almost-simple groups (AS-groups). In the program for computing with non-soluble finite groups, the goal is to reduce the solution of many problems concerning a non-soluble group G to that of solving the same problem for the non-abelian simple composition factors of G. We are concerned with very specific types of computation with AS-groups.

The techniques described in this chapter are under development and are very incomplete in their coverage. The material falls roughly into two main categories.

(a) Functions which try to identify a particular group S known to be almost simple with a standard copy T of that AS-group. In addition, if such an isomorphism is found, it is often desirable to explicitly construct it so that questions concerning S can be answered by mapping them into the "standard" group T. Hence the recognition functions are divided into those which perform *non-constructive recognition* (they assert the existence of an isomorphism between S and T) and those that perform *constructive recognition* (an explicit isomorphism between S and T is returned).

(b) Functions which allow the user to determine information about an AS-group. These functions are usually implemented separately for each family of simple groups. Thus, for each family of simple groups our goal is to provide machinery for constructing key properties of any group T in that family in the context of a standard representation of the group. Using the isomorphism constructed in (a), this information can then be transferred back to the user's group S. Examples of such information include, information about element conjugacy, maximal subgroups, and Sylow p-subgroups.

The functions described in this chapter do not assume that a BSGS-representation can be constructed available. Thus, the techniques described here apply to groups possibly having both much larger order and/or much larger dimension than those that can be handled with the techniques of Chapters 58 and 59.

65.2 Creating Finite Groups of Lie Type

Several functions are provided which construct various classical groups and other groups of Lie type. The effect of these functions is to define the group in terms of a set of generating matrices.

As shown by Chevalley, for each simple Lie algebra L over the complex field and for each finite field $\mathbf{F}_q$ there is an associated matrix group $L(q)$. In general, these groups are perfect but not simple. To obtain the simple group, it is necessary to form the quotient by the centre. Similarly, as Steinberg, Ree and others have shown, if the associated Coxeter graph has an automorphism, of order t say, then there will be a 'twisted' version ${}^tL(q)$ of $L(q)$.

Generators for the series A, C, ^{2}A and ^{2}B are described in [Tay87]. Generators for the series B, D and ^{2}D are as given by Rylands and Taylor [RT98]. Generators for the exceptional groups of Lie type are described by Howlett, Rylands and Taylor in [HRT01].

65.2.1 Generic Creation Function

`ChevalleyGroup(X, n, K:` *parameters*`)`

`ChevalleyGroup(X, n, q:` *parameters*`)`

`Irreducible` BoolElt *Default :* `false`

Construct a matrix group over the field K (or over $\mathbf{F}_q$) which has the adjoint Chevalley group of Lie series `X` and Lie rank n as the quotient modulo scalar matrices. In most cases the group returned is the universal Chevalley group $X_n(q)$; however, for series `B`, `D` and `2D` the universal group is the spin group and the matrix group returned by `ChevalleyGroup` is $\Omega(2n+1,q)$, $\Omega^+(2n,q)$ or $\Omega^-(2n,q)$.

For the twisted groups the meaning of the parameter q is consistent with the (abbreviated) notation in the 'Atlas of Finite Groups' and in the monograph series 'The Classification of the Finite Simple Groups' by Gorenstein, Lyons and Solomon. For a Chevalley group of rank n and type X with an automorphism of order t the Atlas defines the *twisted Chevalley group* ${}^tX_n(q,q^t)$ to be the set of elements of $X_n(q^t)$ fixed by the quotient of the twisting automorphism and the field automorphism induced by $x \mapsto x^q$ of $\mathbf{F}_{q^t}$. In the Atlas the abbreviated notation for the twisted group is ${}^tX_n(q)$ but in Carter [Car72] it is ${}^tX_n(q^t)$. The first signature of the intrinsic expects the field $\mathbf{F}_{q^t}$ but the second signature expects the parameter q.

For example, for the series `"2A"`, the group ${}^2A_n(q)$ is $\mathrm{SU}(n+1,q)$ but, in the first form of the signature, K must be the field $\mathbf{F}_{q^2}$. Similarly the first form of the signature for the groups ${}^3D_4(q)$ and ${}^2E_6(q)$ requires the fields $\mathbf{F}_{q^3}$ and $\mathbf{F}_{q^2}$, respectively.

The possible series and the groups returned are:

`"A"` : $n \geq 0$, $A_n(q)$, the special linear group $\mathrm{SL}(n+1,q)$.

`"B"` : $n \geq 1$, $B_n(q)$, the orthogonal group $\Omega(2n+1,q)$.

`"C"` : $n \geq 1$, $C_n(q)$, the symplectic group $\mathrm{Sp}(2n,q)$.

`"D"` : $n \geq 1$, $D_n(q)$, the orthogonal group $\Omega^+(2n, q)$.

`"E"` : $n \in \{6, 7, 8\}$, the exceptional groups $E_n(q)$. $E_6(q)$ is represented as a matrix group of degree 27. It is simple unless $q \equiv 1 \bmod 3$, in which case its centre has order 3. $E_7(q)$ is represented as a matrix group of degree 56. It is simple unless $q \equiv 1 \bmod 2$, in which case its centre has order 2. $E_8(q)$ is represented as a matrix group of degree 248.

`"F"` : $n = 4$, the exceptional group $F_4(q)$ represented as a matrix group of degree 26. If $q = 3^k$ then this representation is reducible. An irreducible representation is not yet available.

`"G"` : $n = 2$, the exceptional group $G_2(q)$ represented as a matrix group of degree 7. If $q = 2^k$ then this representation is reducible. An irreducible representation of degree 6 can be obtained by setting the parameter `Irreducible := true`.

`"2A"` : $n \geq 1$, $K = \mathbf{F}_{q^2}$, the special unitary group ${}^2A_n(q) = \mathrm{SU}(n+1, q)$.

`"2B"` : $n = 2$, $K = \mathbf{F}_q$, $q = 2^{2k+1}$, the Suzuki group ${}^2B_2(q) = \mathrm{Sz}(q)$.

`"2D"` : $n \geq 1$, $K = \mathbf{F}_q$, ${}^2D_n(q)$, the orthogonal group $\Omega^-(2n, q)$.

`"3D"` : $n = 4$, $K = \mathbf{F}_{q^3}$, the exceptional group ${}^3D_4(q)$.

`"2E"` : $n = 6$, $K = \mathbf{F}_{q^2}$, the exceptional group ${}^2E_6(q)$.

`"2F"` : $n = 4$, $K = \mathbf{F}_q$, $q = 2^{2k+1}$, the Ree group ${}^2F_4(q)$, simple except when $q = 2$ when the derived group is simple and is returned by the function `TitsGroup`.

`"2G"` : $n = 2$, $K = \mathbf{F}_q$, $q = 3^{2k+1}$, the Ree group ${}^2G_2(q)$, simple except when $q = 3$.

65.2.2 The Orders of the Chevalley Groups

`ChevalleyOrderPolynomial(type, n: parameters)`

The orders of the universal Chevalley groups $X_n(q)$ and ${}^tX_n(q)$ are polynomials in q. For the twisted groups of types ${}^2A_n$, ${}^3D_4$ and ${}^2E_6$ the parameter q is the order of the fixed field of the Frobenius automorphism.

Other versions of Chevalley groups are quotients of universal Chevalley groups modulo a subgroup of the centre.

`FactoredChevalleyGroupOrder(type, n, F: parameters)`

`FactoredChevalleyGroupOrder(type, n, q: parameters)`

`Proof`	BoolElt	*Default :* `true`
`Version`	MonStgElt	*Default :* "*Default*"

`ChevalleyGroupOrder(type, n, F: parameters)`

`ChevalleyGroupOrder(type, n, q: parameters)`

`Version` MonStgElt *Default : "Default"*

The (factored) order of the Chevalley group of a given type and rank over the field F (or $\mathbf{F}_q$). The default is the order of the group returned by `ChevalleyGroup`, which except for types B_n, D_n and ${}^2D_n$ is the universal group. The orders of the universal and adjoint Chevalley group can be obtained by setting the parameter `Version` to `Universal` or `Adjoint`. In the factored version the value of `Proof` is passed to the Magma's factorisation function (q.v.).

65.2.3 Classical Groups

Magma offers several functions to construct the classical groups. For most of these functions, it is possible to specify the particular group by giving one of the following combinations of arguments:

(i) The degree n and the coefficient field K of the desired matrix group;

(ii) The degree n of the desired matrix group and a prime power q which relates the group to the appropriate Lie algebra. With the exception of the unitary groups (which will be defined over $\mathbf{F}_{q^2}$), the resulting group will be defined over $\mathbf{F}_q$; or,

(iii) A full vector space $V = K^n$ on which the desired matrix group should act naturally.

65.2.3.1 Linear Groups

`GeneralLinearGroup(n, q)`

`GeneralLinearGroup(n, K)`

`GeneralLinearGroup(V)`

`GL(n, q)`

`GL(n, K)`

`GL(V)`

Here n is a positive integer, q is the power of a prime, K is a finite field $\mathbf{F}_q$, and V is an n-dimensional vector space over K. This function constructs the general linear group $\mathrm{GL}(n, q)$ (resp. $\mathrm{GL}(n, K)$, $\mathrm{GL}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `GL`.

`SpecialLinearGroup(n, q)`

`SpecialLinearGroup(n, K)`

`SpecialLinearGroup(V)`

`SL(n, q)`

`SL(n, K)`

`SL(V)`

Here n is a positive integer, q is the power of a prime, K is a finite field $\mathbf{F}_q$, and V is an n-dimensional vector space over K. This function constructs the special linear group $\mathrm{GL}(n, q)$ (resp. $\mathrm{GL}(n, K)$, $\mathrm{GL}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `SL`.

```
AffineGeneralLinearGroup(GrpMat, n, q)
AffineGeneralLinearGroup(GrpMat, n, K)
AffineGeneralLinearGroup(GrpMat, V)
AGL(GrpMat, V)
```

Here n is a positive integer greater than or equal to 2, q is the power of a prime, K is a finite field $\mathbf{F}_q$, and V is an n-dimensional vector space over K. This function constructs the affine general linear group $\mathrm{AGL}(n, q)$ (resp. $\mathrm{AGL}(n, K)$, $\mathrm{AGL}(V)$) as a subgroup of $\mathrm{GL}(n+1, K)$. If the category name `GrpMat` is omitted the affine group will be returned as a permutation group. The intrinsic name may be abbreviated to `AGL`.

```
AffineSpecialLinearGroup(GrpMat, n, q)
AffineSpecialLinearGroup(GrpMat, n, K)
AffineSpecialLinearGroup(GrpMat, V)
ASL(GrpMat, V)
```

Here n is a positive integer greater than or equal to 2, q is the power of a prime, K is a finite field $\mathbf{F}_q$, and V is an n-dimensional vector space over K. This function constructs the affine special linear group $\mathrm{ASL}(n, q)$ (resp. $\mathrm{ASL}(n, K)$, $\mathrm{ASL}(V)$) as a subgroup of $\mathrm{SL}(n+1, K)$. If the category name `GrpMat` is omitted, the affine group will be returned as a permutation group. The intrinsic name may be abbreviated to `ASL`.

65.2.3.2 Unitary Groups

```
ConformalUnitaryGroup(n, q)
ConformalUnitaryGroup(n, K)
ConformalUnitaryGroup(V)
CU(n, q)
CU(n, K)
CU(V)
```

Here $n \geq 2$ is a positive integer, q is the power of a prime, K is the finite field $\mathbf{F}_{q^2}$, and V is the n-dimensional vector space over K. This function constructs the conformal unitary group $\mathrm{CU}(n, q)$ (resp. $\mathrm{CU}(n, K)$, $\mathrm{CU}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `CU`. A conformal unitary group is the group that preserves a unitary form up to a constant.

```
GeneralUnitaryGroup(n, q)
GeneralUnitaryGroup(n, K)
GeneralUnitaryGroup(V)
GU(n, q)
GU(n, K)
GU(V)
```

Here $n \geq 2$ is a positive integer, q is the power of a prime, K is the finite field $\mathbf{F}_{q^2}$, and V is the n-dimensional vector space over K. This function constructs the general unitary group $\mathrm{GU}(n, q)$ (resp. $\mathrm{GU}(n, K)$, $\mathrm{GU}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `GU`.

```
SpecialUnitaryGroup(n, q)
SpecialUnitaryGroup(n, K)
SpecialUnitaryGroup(V)
SU(n, q)
SU(n, K)
SU(V)
```

Here n is an integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_{q^2}$, and V is the n-dimensional vector space over K. This function constructs the special unitary group $\mathrm{SU}(n, q)$ (resp. $\mathrm{SU}(n, K)$, $\mathrm{SU}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `SU`.

65.2.3.3 Symplectic Groups

```
ConformalSymplecticGroup(n, q)
ConformalSymplecticGroup(n, K)
ConformalSymplecticGroup(V)
CSp(n, q)
CSp(n, K)
CSp(V)
```

Here n is an even integer greater than or equal to 4, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the conformal symplectic group $\mathrm{CSp}(n, q)$ (resp. $\mathrm{CSp}(n, K)$, $\mathrm{CSp}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `CSp`. A conformal symplectic group is the group that preserves a symplectic form up to a constant.

```
SymplecticGroup(n, q)
SymplecticGroup(n, K)
SymplecticGroup(V)
Sp(n, q)
Sp(n, K)
Sp(V)
```

Here n is an even integer greater than or equal to 4, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the symplectic group $\mathrm{Sp}(n, q)$ (resp. $\mathrm{Sp}(n, K)$, $\mathrm{Sp}(V)$) in terms of two generating matrices. The intrinsic name may be abbreviated to `Sp`.

65.2.3.4 Orthogonal and Spin Groups

```
ConformalOrthogonalGroup(n, q)
ConformalOrthogonalGroup(n, K)
ConformalOrthogonalGroup(V)
CO(n, q)
CO(n, K)
CO(V)
```

Here n is an odd integer greater than or equal to 3, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the conformal orthogonal group $\mathrm{CO}(n, q)$ (resp. $\mathrm{CO}(n, K)$, $\mathrm{CO}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `CO`.

```
GeneralOrthogonalGroup(n, q)
GeneralOrthogonalGroup(n, K)
GeneralOrthogonalGroup(V)
GO(n, q)
GO(n, K)
GO(V)
```

Here n is an odd integer greater than or equal to 3, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the general orthogonal group $\mathrm{GO}(n, q)$ (resp. $\mathrm{GO}(n, K)$, $\mathrm{GO}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `GO`.

```
SpecialOrthogonalGroup(n, q)
SpecialOrthogonalGroup(n, K)
```

```
SpecialOrthogonalGroup(V)
SO(n, q)
SO(n, K)
SO(V)
```

Here n is an odd integer greater than or equal to 3, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the special orthogonal group $\mathrm{SO}(n, q)$ (resp. $\mathrm{SO}(n, K)$, $\mathrm{SO}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `SO`.

```
ConformalOrthogonalGroupPlus(n, q)
ConformalOrthogonalGroupPlus(n, K)
ConformalOrthogonalGroupPlus(V)
COPlus(n, q)
COPlus(n, K)
COPlus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the conformal orthogonal group $\mathrm{CO}^+(n, q)$ (resp. $\mathrm{CO}^+(n, K)$, $\mathrm{CO}^+(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `COPlus`.

```
GeneralOrthogonalGroupPlus(n, q)
GeneralOrthogonalGroupPlus(n, K)
GeneralOrthogonalGroupPlus(V)
GOPlus(n, q)
GOPlus(n, K)
GOPlus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the general orthogonal group $\mathrm{GO}^+(n, q)$ (resp. $\mathrm{GO}^+(n, K)$, $\mathrm{GO}^+(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `GOPlus`.

```
SpecialOrthogonalGroupPlus(n, q)
SpecialOrthogonalGroupPlus(n, K)
SpecialOrthogonalGroupPlus(V)
SOPlus(n, q)
SOPlus(n, K)
SOPlus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the special orthogonal group $\mathrm{SO}^+(n,q)$ (resp. $\mathrm{SO}^+(n,K)$, $\mathrm{SO}^+(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `SOPlus`.

```
ConformalOrthogonalGroupMinus(n, q)
ConformalOrthogonalGroupMinus(n, K)
ConformalOrthogonalGroupMinus(V)
COMinus(n, q)
COMinus(n, K)
COMinus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the conformal orthogonal group $\mathrm{CO}^-(n,q)$ (resp. $\mathrm{CO}^-(n,K)$, $\mathrm{CO}^-(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `COMinus`.

```
GeneralOrthogonalGroupMinus(n, q)
GeneralOrthogonalGroupMinus(n, K)
GeneralOrthogonalGroupMinus(V)
GOMinus(n, q)
GOMinus(n, K)
GOMinus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the general orthogonal group $\mathrm{GO}^-(n,q)$ (resp. $\mathrm{GO}^-(n,K)$, $\mathrm{GO}^-(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `GOMinus`.

```
SpecialOrthogonalGroupMinus(n, q)
SpecialOrthogonalGroupMinus(n, K)
SpecialOrthogonalGroupMinus(V)
SOMinus(n, q)
SOMinus(n, K)
SOMinus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the special orthogonal group $\mathrm{SO}^-(n,q)$ (resp. $\mathrm{SO}^-(n,K)$, $\mathrm{SO}^-(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `SOMinus`.

```
Omega(n, q)
```

`Omega(n, K)`

`Omega(V)`

Here n is an odd integer greater than or equal to 3, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the orthogonal group $\Omega(n, K)$ (resp. $\Omega(n, K)$, $\Omega(V)$) in terms of two generating matrices. The group $\Omega(n, K)$ is the kernel of the spinor norm map on $\mathrm{SO}(n, K)$).

`OmegaPlus(n, q)`

`OmegaPlus(n, K)`

`OmegaPlus(V)`

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the orthogonal group $\Omega^+(n, q)$ (resp. $\Omega^+(n, K)$, $\Omega^+(V)$) in terms of two generating matrices. The group $\Omega^+(n, K)$ is the kernel of the spinor norm map on $\mathrm{SO}^+(n, K)$.

`OmegaMinus(n, q)`

`OmegaMinus(n, K)`

`OmegaMinus(V)`

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the orthogonal group $\Omega^-(n, q)$ (resp. $\Omega^-(n, K)$, $\Omega^-(V)$) in terms of two generating matrices. The group $\Omega^-(n, K)$ is the kernel of the spinor norm map on $\mathrm{SO}^-(n, K)$.

`Spin(n, q)`

`Spin(n, K)`

`Spin(V)`

Here n is an odd integer greater than or equal to 1, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the spin group $\mathrm{Spin}(n, K)$ (resp. $\mathrm{Spin}(n, K)$, $\mathrm{Spin}(V)$).

`SpinPlus(n, q)`

`SpinPlus(n, K)`

`SpinPlus(V)`

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the spin group $\mathrm{Spin}^+(n, K)$ (resp. $\mathrm{Spin}^+(n, K)$, $\mathrm{Spin}^+(V)$).

```
SpinMinus(n, q)
SpinMinus(n, K)
SpinMinus(V)
```

Here n is an even integer greater than or equal to 4, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the spin group $\mathrm{Spin}^-(n, K)$ (resp. $\mathrm{Spin}^-(n, K)$, $\mathrm{Spin}^-(V)$).

65.2.4 Exceptional Groups

65.2.4.1 Suzuki Groups

The Suzuki groups are specified slightly differently, as the degree of the group is always four. Thus for this family of groups, the possible combinations of arguments are:

(i) A finite field $K = \mathbf{F}_{2^{2m+1}}$, over which the resulting matrix group is defined;

(ii) An integer $q = 2^{2m+1}$, corresponding to the field $K = \mathbf{F}_q$ over which the resulting matrix group is defined; or,

(iii) A vector space $V = K^4$ where $K = \mathbf{F}_{2^{2m+1}}$ on which the resulting matrix group acts naturally. which the resulting

```
SuzukiGroup(q)
SuzukiGroup(K)
SuzukiGroup(V)
```

Here q is a prime power of the form 2^{2n+1}, K is the finite field $\mathbf{F}_q$, and V is the 4-dimensional vector space over K. This function constructs the Suzuki simple group $\mathrm{Sz}(q)$ (resp. $\mathrm{Sz}(K)$, $\mathrm{Sz}(V)$) in terms of two generating matrices. The intrinsic name may be abbreviated to `Sz`.

Example H65E1

We create the 10-dimensional symplectic group over $\mathbf{F}_8$:

```
> F<u> := FiniteField(8);
> G := SymplecticGroup(10, F);
> G;
MatrixGroup(10, GF(2, 3))
Generators:
[  u   0   0   0   0   0   0   0   0   0]
[  0   1   0   0   0   0   0   0   0   0]
[  0   0   1   0   0   0   0   0   0   0]
[  0   0   0   1   0   0   0   0   0   0]
[  0   0   0   0   u   0   0   0   0   0]
[  0   0   0   0   0   u   0   0   0   0]
[  0   0   0   0   0   0   1   0   0   0]
[  0   0   0   0   0   0   0   1   0   0]
```

```
[  0   0   0   0   0   0   0   0   1   0]
[  0   0   0   0   0   0   0   0   0 u^6]

[0 0 0 1 1 1 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 1 0 1 0 0 0]
[0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 1 0 0 0 0 0]
```

Example H65E2

We create the Suzuki group over $\mathbf{F}_{128}$:

```
> F<w> := FiniteField(128);
> V := VectorSpace(F, 4);
> S := SuzukiGroup(V);
> S;
MatrixGroup(4, GF(2, 7))
Generators:
[0 0 0 1]
[0 0 1 0]
[0 1 0 0]
[1 0 0 0]

[  w^8     0     0     0]
[    0 w^120     0     0]
[    0     0   w^7     0]
[    0     0     0 w^119]

[   1    0    0    0]
[ w^8    1    0    0]
[   0    w    1    0]
[w^17  w^9  w^8    1]
> Order(S);
34093383680
> FactoredOrder(S);
[ <2, 14>, <5, 1>, <29, 1>, <113, 1>, <127, 1> ]
```

65.2.4.2 Small Ree Groups

The Ree groups (${}^2G_2(q)$) are given in an irreducible matrix representation of degree seven. The possible combinations of arguments are:

(i) A finite field $K = \mathbf{F}_{3^{2m+1}}$ with $m > 0$, over which the matrix group is defined.

(ii) An integer $q = 3^{2m+1}$ with $m > 0$, corresponding to the field $K = \mathbf{F}_q$ over which the group is defined; or,

(iii) A vector space $V = K^7$ where $K = \mathbf{F}_{3^{2m+1}}$ with $m > 0$, on which the matrix group acts naturally.

```
ReeGroup(q)
ReeGroup(K)
ReeGroup(V)
```

Here q is a prime power of the form $q = 3^{2m+1}$ with $m > 0$, K is the finite field $\mathbf{F}_q$, and V is the 7-dimensional vector space over K. This function constructs the Ree group ${}^2G_2(q)$ (resp. ${}^2G_2(K)$, ${}^2G_2(V)$) in terms of standard generating matrices. The intrinsic name may be abbreviated to `Ree`.

65.2.4.3 Large Ree Groups

The Ree groups (${}^2F_4(q)$) are given in an irreducible matrix representation of degree twenty-six. The possible combinations of arguments are:

(i) A finite field $K = \mathbf{F}_{2^{2m+1}}$ with $m > 0$, over which the matrix group is defined.

(ii) An integer $q = 2^{2m+1}$ with $m > 0$, corresponding to the field $K = \mathbf{F}_q$ over which the group is defined; or,

(iii) A vector space $V = K^{26}$ where $K = \mathbf{F}_{2^{2m+1}}$ with $m > 0$, on which the matrix group acts naturally.

```
LargeReeGroup(q)
LargeReeGroup(K)
LargeReeGroup(V)
```

Here q is a prime power of the form $q = 2^{2m+1}$ with $m > 0$, K is the finite field $\mathbf{F}_q$, and V is the 26-dimensional vector space over K. This function constructs the Ree group ${}^2F_4(q)$ (resp. ${}^2F_4(K)$, ${}^2F_4(V)$) in terms of standard generating matrices. The intrinsic name may be abbreviated to `LargeRee`.

65.3 Group Recognition

65.3.1 Constructive Recognition of Alternating Groups

`RecogniseAlternatingOrSymmetric(G, n)`

Constructive recognition of the group G, which will succeed with probability $\geq 1-e^5$ if G is isomorphic to either the alternating or symmetric group of degree $n > 11$. The method is that of Beals *et al* [BLGN$^+$03], implemented by Colva Roney-Dougal.

The return values start with a flag indicating success or failure. If the algorithm was successful, then there are three more return values: a flag which is true when G is symmetric and false when alternating, and two programs. The first program takes an element x of an overgroup of G and produces a boolean to indicate whether $x \in G$ and a permutation representing x in the natural action of S_n (if such a permutation exists). The second taking a permutation to the corresponding element of G. The programs define mutually inverse group isomorphisms, implemented as MAGMA functions.

Example H65E3

We give an example of `RecogniseAlternatingOrSymmetric` in use.

```
> a:= AlternatingGroup(13);
> h:= Stabiliser(a, {1,2});
> k:= CosetImage(a, h);
> Degree(k);
78
> worked, is_sym, bb_to_perm, perm_to_bb:=
> RecogniseAlternatingOrSymmetric(k, 13);
> worked;
true
> is_sym;
false
> x:= Sym(78)!(1, 35, 16, 28, 14, 26, 69, 5, 74)(2, 54,
> 67, 18, 51, 63, 6, 50, 77)(3, 33, 78, 12, 34, 29, 19, 15, 73)
> (4, 52, 61, 24, 49, 60, 68, 38, 64)(7, 20, 71, 17,
> 32, 11, 72, 8, 36)(9, 76, 47, 31, 56, 62, 13, 53, 59)
> (10, 70, 57, 23, 37, 22, 21, 27, 25)(30, 45, 46, 43, 42,
> 44, 40, 41, 75)(39, 55, 65)(48, 66, 58);
> x in k;
true;
> in_k, perm_image:= bb_to_perm(x);
> in_k;
true
> perm_image;
(1, 2, 3)(4, 7, 12, 6, 10, 11, 13, 9, 8)
> perm_to_bb(perm_image) eq x;
true
```

`RecogniseSymmetric(G, n: parameters)`

`maxtries`	RngIntElt	*Default :* $100n + 5000$
`Extension`	BoolElt	*Default :* `false`

The group G should be known to be isomorphic to the symmetric group S_n for some $n \geq 8$. The Bratus-Pak algorithm [BP00] (implemented by Derek Holt) is used to define an isomorphism between G and S_n. If successful, return `true`, homomorphism from G to S_n, homomorphism from S_n to G, the map from G to its word group and the map from the word group to G.

If the optional parameter `Extension` is set, then the group G should be known to be isomorphic either to S_n or to a perfect central extension $2.S_n$. In that case, the first two maps returned will be a homomorphism from G to S_n and a map from S_n to G that induces a homomorphism onto $G/Z(G)$. The sixth value returned will be `true`, if $G \cong 2.S_n$ and `false`, if $G \cong 2.A_n$.

If unsuccessful, `false` is returned. This will always occur if the input group is not isomorphic to S_n (or $2.S_n$ when `Extension` is set) with $n \geq 8$, and may occur occasionally even when G is isomorphic to S_n. The optional parameter `maxtries` (default $100n + 5000$) can be used to control the number of random elements chosen before giving up.

`SymmetricElementToWord (G, g)`

If g is an element of G which has been constructively recognised to be isomorphic to S_n (or $2.S_n$), then return `true` and element of word group for G which evaluates to g. Otherwise return `false`. This facilitates membership testing in G.

`RecogniseAlternating(G, n: parameters)`

`maxtries`	RngIntElt	*Default :* $100n + 5000$
`Extension`	BoolElt	*Default :* `false`

The group G should be known to be isomorphic to the alternating group A_n for some $n \geq 9$. The Bratus-Pak algorithm [BP00] (implemented by Derek Holt) is used to define an isomorphism between G and A_n. If successful, return `true`, homomorphism from G to A_n, homomorphism from A_n to G, the map from G to its word group and the map from the word group to G.

If the optional parameter `Extension` is set, then the group G should be known to be isomorphic either to A_n or to a perfect central extension $2.A_n$. In that case, the first two maps returned will be a homomorphism from G to A_n and a map from A_n to G that induces a homomorphism onto $G/Z(G)$. The sixth value returned will be `true`, if $G \cong 2.A_n$ and `false`, if $G \cong 2.A_n$.

If unsuccessful, `false` is returned. This will always occur if the input group is not isomorphic to A_n (or $2.A_n$ when `Extension` is set) with $n \geq 9$, and may occur occasionally even when G is isomorphic to A_n. The optional parameter tt maxtries (default $100n + 5000$) can be used to control the number of random elements chosen before giving up.

`AlternatingElementToWord (G, g)`

If g is an element of G which has been constructively recognised to be isomorphic to A_n (or $2.A_n$), then return `true` and element of word group for G which evaluates to g. Otherwise return `false`. This facilitates membership testing in G.

`GuessAltsymDegree(G: parameters)`

`maxtries`	RngIntElt	*Default :* 5000
`Extension`	BoolElt	*Default :* `false`

The group G should be believed to be isomorphic to S_n or A_n for some $n > 6$, or to $2.S_n$ or $2.A_n$ if the optional parameter `Extension` is set. This function attempts to determine n and whether G is symmetric or alternating. It does this by sampling orders of elements. It returns either `false`, if it is unable to make a decision after sampling `maxtries` elements (default 5000), or `true`, *type* and n, where *type* is "Symmetric" or "Alternating", and n is the degree. If G is not isomorphic to S_n or A_n (or $2.S_n$ or $2.A_n$ when `Extension` is set) for $n > 6$, then the output is meaningless - there is no guarantee that `false` will be returned. There is also a small probability of a wrong result or `false` being returned even when G is S_n or A_n with $n > 6$. This function was written by Derek Holt.

Example H65E4

For a group G which is believed to be isomorphic to S_n or A_n for some unknown value of $n > 6$, the function `GuessAltsymDegree` can be used to try to guess n, and then `RecogniseSymmetric` or `RecogniseAlternating` can be used to confirm the guess.

```
> G:= sub< GL(10,5) |
> PermutationMatrix(GF(5),Sym(10)![2,3,4,5,6,7,8,9,1,10]),
> PermutationMatrix(GF(5),Sym(10)![1,3,4,5,6,7,8,9,10,2]) >;
> GuessAltsymDegree(G);
true Alternating 10
> flag, m1, m2, m3, m4  := RecogniseAlternating(G,10);
> flag;
true
> x:=Random(G); Order(x);
8
> m1(x);
(1, 2, 4, 9, 10, 8, 6, 3)(5, 7)
> m2(m1(x)) eq x;
true
> m4(m3(x)) eq x;
true
> flag, w := AlternatingElementToWord(G,x);
> flag;
true
> m4(w) eq x;
true
```

```
> y := Random(Generic(G));
> flag, w := AlternatingElementToWord(G,y);
> flag;
false
> flag, m1, m2, m3, m4 := RecogniseAlternating(G,11);
> flag;
false
> flag, m1, m2, m3, m4 := RecogniseSymmetric(G,10);
> flag;
false
```

The nature of the `GuessAltsymDegree` function is that it assumes that its input is either an alternating or symmetric group and then tries to guess which one and the degree. As such, it is almost always correct when the input is an alternating or symmetric group, but will often return a bad guess when the input group is not of this form, as in the following example.

```
> GuessAltsymDegree(Sym(50));
true Symmetric 50
> GuessAltsymDegree(Alt(73));
true Alternating 73
> GuessAltsymDegree(PSL(5,5));
true Alternating 82
```

65.3.2 Determining the Type of a Finite Group of Lie Type

Given a finite quasisimple group of Lie type in any representation, the functions in this section apply probabilistic algorithms to determine its defining characteristic and type as a Lie group.

`LieCharacteristic(G : parameters)`

`NumberRandom`	RngIntElt	*Default :* 100
`Verify`	BoolElt	*Default :* `true`

Given a finite quasisimple permutation or matrix group G which is of Lie type, determine its defining characteristic. The Monte Carlo algorithm implemented by this function is that of Liebeck and O'Brien [LO07]. Since it is Monte Carlo, there is a small probability of error. The number of random elements considered is `NumberRandom`. If `Verify` is `true`, then we first verify that G is perfect by applying `IsProbablyPerfect`.

Example H65E5

```
> F := GF (4);
> w := PrimitiveElement (F);
> a := [
> 0,w^3,0,0,0,
> w^3,0,0,0,0,
```

```
> 0,0,0,w^3,0,
> 0,0,w^3,0,0,
> w^2,w^2,w^3,w^3,w^3];
> b := [
> 0,0,w^3,0,0,
> w^1,w^2,w^2,0,0,
> w^2,w^1,w^2,0,0,
> 0,0,0,0,w^3,
> w^2,w^2,w^2,w^3,w^3];
> G := sub <GL(5, F) | a, b>;
> LieCharacteristic(G);
11
```

LieType(G, p : *parameters*)

LieType(G, p : *parameters*)

NumberRandom RngIntElt *Default :* 100

If the matrix or permutation group G is nearly simple, and its non-abelian composition factor is isomorphic to a group of Lie type in characteristic p, then this function returns `true` and its standard Chevalley name. Otherwise it returns `false`.

The algorithm is that of Babai, Kantor, Pálfy and Seress [BKPS02]; this implementation was developed by Malle and O'Brien. Since it is Monte Carlo, there is a small probability of error. The number of random elements considered is `NumberRandom`.

The standard name is a tuple that defines the isomorphism type of the composition factor. It is similar to that employed by `CompositionFactors`, described in the Permutation Groups chapter.

If the composition factor is a group of Lie type, then the tuple is $< s, n, q >$ and it defines the adjoint Chevalley group of Lie series s and Lie rank n over $GF(q)$. The tuple entries are valid arguments for `ChevalleyGroup`.

If the composition factor is an alternating group, and so lies in family 17, then the tuple is $< 17, n, 0 >$ and it defines the alternating group of degree n.

If the composition factor is a sporadic group and so lies in family 18, then the tuple is $< 18, n, s >$; the string s is its standard Atlas name and n is the number of the group in family 18.

SimpleGroupName(G : *parameters*)

SimpleGroupName(G : *parameters*)

NumberRandom RngIntElt *Default :* 100

If the matrix or permutation group G is nearly simple, this function returns `true` and a list of possible names for its non-abelian simple composition factor; otherwise it returns `false`. Since it is Monte Carlo, there is a small probability of error.

The number of random elements considered is `NumberRandom`. The list of standard names follows the convention described above.

The algorithm and implementation were developed by Malle and O'Brien; it uses `LieType` and `LieCharacteristic`.

Example H65E6

We create the classical group $\Omega(7,5)$ in its natural representation and apply `SimpleGroupName` to it.

```
> SetSeed(1);
> G := Omega(7, 5);
> flag, name := SimpleGroupName(G);
> name;
[* <B, 3, 5> *]
```

We create a certain 5-dimensional matrix group over $GF(3)$ and determine which simple group it is.

```
> F := GF(3);
> P := GL(5,F);
> gens := [
> P![2,1,2,1,2,2,0,0,0,2,0,2,0,0,0,0,1,2,0,1,1,0,2,2,1],
> P![2,1,0,2,1,1,2,0,2,2,1,1,2,1,1,0,2,0,1,1,1,1,2,2,2]];
> G := sub <P | gens>;
> flag, name := SimpleGroupName(G);
> flag;
true
> name;
[* <18, 1, M11> *]
```

¿ /* naming an alternating group */

```
> G := MatrixGroup<4, GF(2) |
>    [ 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0 ],
>    [ 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0 ] >;
> flag, name := SimpleGroupName(G);
> flag;
true
> /* this is A5 */
> name;
[* <17, 5, 0> *]
> /* naming a classical group */
> F := GF(7^2);
> P := GL (6,F);
> w := PrimitiveElement (F);
> gens := [
> P![w^12,w^36,  0,  5,  2,  0,w^44,w^36,  0,  6,  2,  0,
> w^42,w^42,w^28,w^22,w^22,  3,  4,  3,  0,w^36,w^12,  0,
> 2,  3,  0,w^20,w^12,  0,w^14,w^14,  1,w^18,w^18, w^4],
```

```
>
> P![w^38,w^26,w^25,w^21, w^9,  3,w^21,w^45,w^33, w^4,w^28,
> 2,  6,  4, w^1, w^7,w^15,  4,  1,w^36,w^35, w^5,w^41,  5,
> w^31, w^7,w^43,w^36,w^12,  1,w^34,w^42,w^11,w^39,w^47,  2]
> ];
> G := sub <P | gens>;
> flag, name := LieType(G, 5);
> flag;
true
> name;
<A, 1, 5>
> /* so this is SL(2, 5) */
```

65.3.3 Classical Forms

Let G be an absolutely irreducible subgroup of $\mathrm{GL}(d, q)$. The following functions compute symplectic, unitary and orthogonal forms of the underlying vector space V left invariant by the action of G.

A *bilinear* form is a bilinear function κ from $V \times V \to F$. It is G-invariant modulo scalars if for each $g \in G$ there is a $\mu_g \in F$ such that $\kappa(vg, wg) = \mu_g \kappa(v, w)$ for all $v, w \in V$.

Now suppose that $a \mapsto \bar{a}$ is an automorphism of F of order 2. A *sesquilinear* form is a biadditive function κ from $V \times V \to F$ such that $\kappa(au, bv) = a\bar{b}\kappa(u, v)$ for all $u, v \in V$ and $a, b \in F$. It is G-invariant modulo scalars if for each $g \in G$ there is a $\mu_g \in F$ such that $\kappa(vg, wg) = \mu_g \kappa(v, w)$ for all $v, w \in V$.

A quadratic form is a function $\chi : V \to F$ such that

(1) $\chi(av) = a^2\chi(v)$ for all $a \in F$, $v \in V$; and

(2) the form κ, defined by $\kappa(u, v) = \chi(u + v) - \chi(u) - \chi(v)$ for all $u, v \in V$, is bilinear.

It is G-invariant if for each $g \in G$, $\chi(vg) = \chi(v)$ for all $v \in V$. It is G-invariant modulo scalars if for each $g \in G$ there is a $\mu_g \in F$ such that $\chi(vg) = \mu_g \chi(v)$ for all $v \in V$.

A bilinear form which is G-invariant (modulo scalars) is represented by a matrix B such that $g * B * g^{tr} = \mu_g B$ for all $g \in G$ and is unique up to multiplication by an element of F. Assume F has an automorphism $a \mapsto \bar{a}$ of order 2; a sesquilinear form is a matrix B such that $g * B * \bar{g}^{tr} = \mu_g B$ for all $g \in G$ and is unique up to multiplication by an element of F (where $\bar{g}$ denotes the matrix obtained from g by replacing each entry g_{ij} by $\bar{g_{ij}}$). A quadratic form is represented by an upper triangular matrix Q such that the matrix $g * Q * g^{tr}$, normalized into an upper triangular matrix, equals $\mu_g Q$.

The functions below will exit with an error message if the input group G is reducible. They may also exit with error if G is not absolutely irreducible, or if `Scalars` is `true` and the derived subgroup $[G, G]$ of G is not absolutely irreducible. They may however sometimes succeed in finding a fixed form when G is irreducible but not absolutely irreducible.

`ClassicalForms(G: parameters)`

`Scalars` BoolElt *Default :* `false`

Given as input a matrix group G acting absolutely irreducibly on the underlying vector space V over the field F, `ClassicalForms` will try to find a classical form which is G-invariant or prove that no such form exists. If the optional argument `Scalars` is `true` then it will look for a form which is G-invariant modulo scalars. When `Scalars` is `true`, it is only guaranteed to succeed when $[G, G]$ acts absolutely irreducibly on V. If it finds a fixed form, then it will stop and will not look for alternative fixed forms of different types.

The classical forms are: *symplectic* (non-degenerate, alternating bilinear), *unitary* (non-degenerate sesquilinear) or *orthogonal* (a *symmetric bilinear form* and a *quadratic* form).

The function `ClassicalForms` returns a record `forms` which contains the components `formType`, `sign`, `bilinearForm`, `sesquilinearForm`, `quadraticForm` and `scalars`. Depending on the entry `formType` the record components are set to indicate:

`"unknown"`	: it is not known whether G fixes a classical form.
`"linear"`	: it is known that G does not fix a classical form modulo scalars.
`"symplectic"`	: G fixes a symplectic form modulo scalars. The matrix of the form is stored in `bilinearForm` and the scalars for each generator of G are stored in `scalars`. In characteristic two this also implies that no quadratic form is fixed.
`"unitary"`	: G fixes a unitary form (modulo scalars). The matrix of the form is stored in `sesquilinearForm`. The scalars for each generator of G are stored in `scalars`.
`"orthogonalcircle"`	:
`"orthogonalplus"`	:
`"orthogonalminus"`	: G fixes an orthogonal form modulo scalars. The matrix of the bilinear form is stored in `bilinearForm` and the corresponding quadratic form in `quadraticForm`. The scalars for each generator of G are stored in `scalars`. In the orthogonal case, `sign` is set to 0, 1, or -1 when `formType` is `"orthogonalcircle"`, `"orthogonalplus"`, or `"orthogonalminus"`, respectively.

`SymplecticForm(G: parameters)`

`Scalars` BoolElt *Default :* `false`

If the absolutely irreducible group G preserves a symplectic form (modulo scalars if the optional argument `Scalars` is `true`), this function returns `true` and the matrix of the form. If it is known that G does not preserve such a form it returns `false`.

If it cannot decide (perhaps because the group does not act absolutely irreducibly), then it exits with an error message. If `Scalars` is `true`, then the list of scalars for the generators of G is also returned.

`SymmetricBilinearForm(G: parameters)`

`Scalars` BoolElt *Default :* `false`

If the absolutely irreducible group G preserves an orthogonal form (modulo scalars if the optional argument `Scalars` is `true`), then this function returns `true`, the matrix of the symmetric bilinear form, and the type of the form (as in `ClassicalForms`). If it is known that G does not preserve such a form, it returns `false`. If it cannot decide, then it exits with an error message. If `Scalars` is `true`, then the list of scalars for the generators of G is also returned.

`QuadraticForm(G)`

`Scalars` BoolElt *Default :* `false`

If the absolutely irreducible group G preserves a quadratic form (modulo scalars if the optional argument `Scalars` is `true`), this function returns `true`, the matrix of the form in upper triangular form, and the type of the form (as in `ClassicalForms`). If it is known that G does not preserve such a form it returns `false`. If it cannot decide, then it exits with an error message. If `Scalars` is `true`, then the list of scalars for the generators of G is also returned.

`UnitaryForm(G)`

`Scalars` BoolElt *Default :* `false`

If the absolutely irreducible group G preserves a unitary form (non-degenerate sesquilinear) (modulo scalars if the optional argument `Scalars` is `true`), then this function returns `true` and the matrix of the form. If it is known that G does not preserve such a form, it returns false. If it cannot decide, then it exits with an error message. If `Scalars` is `true`, then the list of scalars for the generators of G is also returned.

`FormType(G)`

`Scalars` BoolElt *Default :* `false`

If the absolutely irreducible group G preserves a classical form (modulo scalars if the optional argument `Scalars` is `true`), this function returns its type (see `ClassicalForms`). Otherwise it returns `"unknown"`.

Example H65E7

```
> G := Omega( 9, 11 );
> ClassicalForms( G );
rec<recformat<bilinearForm, quadraticForm, sesquilinearForm, bilinFlag,
sesquiFlag, scalars, formType, bc, n> | bilinearForm :=
[ 0  0  0  0  0  0  0  0  1]
[ 0  0  0  0  0  0  0  1  0]
[ 0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  1  0  0  0]
[ 0  0  0  0  6  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0]
[ 1  0  0  0  0  0  0  0  0],
quadraticForm :=
[ 0  0  0  0  0  0  0  0  1]
[ 0  0  0  0  0  0  0  1  0]
[ 0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  1  0  0  0]
[ 0  0  0  0  3  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0],
sesquilinearForm := false, bilinFlag := true,  sesquiFlag := false,
scalars := [ 1, 1 ], formType := orthogonalcircle, sign := 0>
> FormType( G );
orthogonalcircle
> SymplecticForm( G );
false
```

`TransformForm(form, type)`

Return a matrix m such that G^m lies in the classical group returned by the MAGMA function `GU`, `Sp`, or `GO(Plus/Minus)`. The argument $form$ should be a classical form of type *type* fixed by an absolutely irreducible subgroup G of $GL(d, q)$. It should be the bilinear or sesquilinear form fixed by G, except when G is orthogonal in characteristic 2, in which case it should be the quadratic form. The argument *type* should be as in the `formType` component of the record returned by `ClassicalForms`; i.e. one of `"symplectic"`, `"unitary"`, `"orthogonalcircle"`, `"orthogonalplus"`, or `"orthogonalminus"`.

`TransformForm(G)`

`Scalars`	BoolElt	*Default :* `false`

This function calls `ClassicalForms` to find a form fixed by the absolutely irreducible subgroup G of $GL(d, q)$. If `Scalars` is `true`, then `ClassicalForms` is called with `Scalars` set to `true`, so that a form fixed module scalars is found. If a form $form$ of type *type* is fixed, then it returns `TransformForm(form, type)`. Otherwise it returns `false`.

`SpinorNorm(g, form)`

The spinor norm of g with respect to the given form . $form$ must be the matrix of an orthogonal form (ie, it must be symmetric and nonsingular), and g an element of the general orthogonal group `GO(Plus/Minus)` fixing that form. Note that the form is ignored in even characteristic, since the spinor norm of g is just equal to the rank modulo 2 of $g - I$ in that case.

65.3.4 Recognizing Classical Groups in their Natural Representation

Let G be an irreducible subgroup of $\mathrm{GL}(d, q)$. The following algorithm is designed to test whether G contains the corresponding classical group Ω and is contained in Δ. Here Ω and Δ are defined as follows:

Case "linear": $\Delta = \mathrm{GL}(d, q)$, $\Omega = \mathrm{SL}(d, q)$

Case "symplectic": $\Delta = \mathrm{GSp}(d, q)$, $\Omega = \mathrm{Sp}(d, q)$

Case "orthogonalplus": $\Delta = \mathrm{GO}^+(d, q)$, $\Omega = \Omega^+(d, q)$

Case "orthogonalminus": $\Delta = \mathrm{GO}^-(d, q)$, $\Omega = \Omega^-(d, q)$

Case "orthogonalcircle":$\Delta = \mathrm{GO}^\circ(d, q)$, $\Omega = \Omega^\circ(d, q)$

Case "unitary": $\Delta = \mathrm{GU}(d, q)$, $\Omega = \mathrm{SU}(d, q)$

`RecognizeClassical( G : parameters)`

`Case`	MonStgElt	*Default :* "*unknown*"
`NumberOfElements`	RngIntElt	*Default :* 25
Verbose	`Classical`	*Maximum :* 3

`RecognizeClassical` takes as input a group G, which is a subgroup of $\mathrm{GL}(d, q)$.

The parameter `Case` is one of `"linear"`, `"symplectic"`, `"orthogonalplus"`, `"orthogonalminus"`, `"orthogonalcircle"`, `"unitary"` or `"unknown"`; if `Case` is supplied, then the algorithm seeks to decide for this case only.

The parameter `NumberOfElements` is the number of random elements selected from G during the execution of the algorithm.

The output of `RecognizeClassical` is either `true`, `false` or `"Does not apply"`. If the algorithm returns `true`, then we know with certainty that G contains Ω and is contained in Δ. Note that the proof of correctness of the algorithm depends on the finite simple group classification. If it returns `false` then either G does not contain

Ω, or G is not contained in Δ, or G is not irreducible, or there is a small chance that G is contained in Δ and contains Ω. More precisely, if the irreducible group G is contained in Δ and really does contain Ω then the probability with which the algorithm returns `false` is less than ε, where ε is a real number between 0 and 1. The smaller the value of ε, the larger `NumberOfElements` must be. If the algorithm returns `"Does not apply"` then it is not applicable to the given group.

If `"Classical"` is set to verbose then, where `RecognizeClassical` returns `true`, it also prints the statement "Proved that the group contains a classical group of type *case* in n random selections", where n is the number of selections needed. If it returns `false`, it prints a statement giving some indication of why the algorithm reached this conclusion.

Theoretical details of the algorithms used may be found in Niemeyer & Praeger [NP97][NP98][NP99] and Praeger [Pra99]. Its approach is based on the SL-recognition algorithm (Neumann & Praeger, [NP92]).This implementation also uses algorithms described in Celler & Leedham-Green [CLG97][CLG97b] and Celler et al. [CLGM$^+$95].

For small fields ($q < 2^{16}$), the cost of this implementation for a given value of `NumberOfElements` is $O(d^3 \log d)$ bit operations.

`IsLinearGroup(G)`

This function tests whether the subgroup G of $\mathrm{GL}(d, q)$ contains $\mathrm{SL}(d, q)$. If the function can establish this fact, it returns `true` and otherwise `false`. Hence, if `IsLinearGroup` returns `false`, there is a small chance that G nevertheless contains $\mathrm{SL}(d, q)$. See `RecognizeClassical` for more details.

`IsSymplecticGroup(G)`

This function tests whether the subgroup G of $\mathrm{GSp}(d, q)$ contains $\mathrm{Sp}(d, q)$. If the function can establish this fact, it returns `true` and otherwise `false`. Hence, if `IsSymplecticGroup` returns `false`, there is a small chance that G nevertheless contains $\mathrm{Sp}(d, q)$. See `RecognizeClassical` for more details.

`IsOrthogonalGroup(G)`

This function tests whether the subgroup G of $\mathrm{GO}^\epsilon(d, q)$ contains $\Omega^\epsilon(d, q)$. If the function can establish this fact, it returns `true` and otherwise `false`. Hence, if `IsOrthogonalGroup` returns `false`, there is a small chance that G nevertheless contains $\Omega^\epsilon(d, q)$. See `RecognizeClassical` for more details.

`IsUnitaryGroup(G)`

This function tests whether the subgroup G of $\mathrm{GU}(d, q)$ contains $\mathrm{SU}(d, q)$. If the function can establish this fact, it returns `true` and otherwise `false`.

ClassicalType(G)

If G is known to be a classical subgroup of $\mathrm{GL}(d, q)$ this function returns the appropriate classical type as a string, i.e. `"linear"`, `"symplectic"`, `"orthogonalplus"`, `"orthogonalminus"`, `"orthogonalcircle"`, or `"unitary"`. Otherwise the function returns `false`.

Example H65E8

```
> G := SU (60, 9);
> SetVerbose( "Classical", true );
> RecognizeClassical( G );
true
> IsLinearGroup( G );
false
> IsUnitaryGroup( G );
true
> IsSymplecticGroup( G );
false
> IsOrthogonalGroup( G );
false
> ClassicalType( G );
unitary
> G := Sp (462, 3);
> time RecognizeClassical( G );
true
Time: 7.630
```

65.3.5 Constructive Recognition of Linear Groups

The functions in this section recognise whether of not a given group G is a specified linear group T. If it is, then an isomorphism between G and T is returned.

RecognizeSL2(G)

RecognizeSL2(G)

RecognizeSL2(G, q)

RecognizeSL2(G, q)

If G, a matrix or permutation group, is isomorphic, possibly modulo scalars, to $(P)SL(2, q)$, then homomorphisms between G and $(P)SL(2, q)$ are constructed. The function returns a homomorphism from G to $(P)SL(2, q)$, a homomorphism from $(P)SL(2, q)$ to G, the map from G to its word group, and the map from the word group to G.

If q, the cardinality of the defining field for G, is known, it *should* be supplied. Otherwise, the function `SL2Characteristic` is used to determine q; if q is large, this calculation may be **expensive**.

SL2ElementToWord(G, g)

SL2ElementToWord(G, g)

If g is an element of the matrix or permutation group G which has been constructively recognised to have central quotient isomorphic to $PSL(2, q)$, then return `true` and element of word group for G which evaluates to g, else `false`. This facilitates membership testing in G.

SL2Characteristic(G : *parameters*)

SL2Characteristic(G : *parameters*)

NumberRandom	RngIntElt	*Default :* 100
Verify	BoolElt	*Default :* `true`

Subject to the *assumption* that the group G has central quotient $(P)SL(2, q)$, determine its characteristic and field size. The Monte Carlo algorithm implemented by this function is that of Liebeck and O'Brien [LO07]. Since it is Monte Carlo, there is a small probability of error. The number of random elements considered is `NumberRandom`. If `Verify` is `true`, then we first verify that G is perfect by applying `IsProbablyPerfect`.

The constructive recognition algorithms for $SL(2, q)$ were developed by Conder, Leedham-Green and O'Brien [CLGO06]. The algorithm used for other representations was developed by Brooksbank and O'Brien.

Example H65E9

Our first example uses $G = \mathrm{SL}(2, 3^2)$ in its natural representation. We first recognise the group and then express a random matrix of G as a word in the generators of G.

```
> G := SL(2, 3^2);
> flag, phi, tau, gamma, delta := RecogniseSL2(G, 3^2);
> g := G![1, 2, 0, 1];
> w := gamma(g);
> delta(w) eq g;
true
```

Example H65E10

We now consider a representation of a 2-dimensional linear group inside $GL(6, \mathbf{F}_{57})$.

```
> K<w> := GF(5, 7);
> G :=
> MatrixGroup<6, GF(5, 7) |
> [w^19035, w^14713, w^50617, w^14957, w^51504, w^48397, w^16317, w^3829,
>  w^35189, w^2473, w^19497, w^77192, w^46480, w^6772, w^29577, w^61815,
>  w^54313, w^16757, w^43765, w^64406, w^58788, w^30789, w^13579, w^66728,
>  w^7733, w^45434, w^42411, w^61613, w^12905, w^6889, w^50116, w^16117,
>  w^56717, w^25226, w^49940, w^36836 ],
```

```
> [w^63955, w^40568, w^45004, w^11642, w^39536, w^11836, w^52594, w^71166,
>  w^47015, w^74450, w^32373, w^37021, w^76381, w^18155, w^57943, w^31194,
>  w^62524, w^65864, w^11868, w^76867, w^26483, w^41335, w^64856, w^41125,
>  w^43990, w^40104, w^24842, w^3153, w^23777, w^60024, w^14454, w^68648,
>  w^43403, w^26710, w^39779, w^22074 ] >;
>
> flag, phi, tau, gamma, delta := RecogniseSL2(G, 5^7);
> phi;
Mapping from: GrpMat: G to SL(2, GF(5, 7)) given by a rule [no inverse]
> g := Random(G);
> h := phi (g);
> h;
[w^40430   w^970]
[ w^5607 w^11606]
> k := tau(h);
> w := gamma(k);
> m := delta(w);
```

Recall that we are working modulo scalars.

```
> IsScalar(m * g^-1);
true
> H := SL(2, 5^7);
> h := H![1,1,0,1];
> g := tau(h);
> Order(g);
5
```

We now test a random element of $GL(6, \mathbf{F}_{57})$ for membership of our group.

```
> g := Random(GL(6, 5^7));
> SL2ElementToWord(G, g);
false
```

`RecogniseSL3(G)`

`RecogniseSL3(G, q : parameters)`

`Verify` BoolElt *Default :* `true`

If $G \leq GL(d, F)$, is isomorphic, possibly modulo scalars, to $(P)SL(3, q)$, then construct homomorphisms between G and $(P)SL(3, q)$. Return homomorphism from G to $(P)SL(3, q)$, homomorphism from $(P)SL(3, q)$ to G, the map from G to its word group and the map from the word group to G.

If q, the cardinality of the defining field for G, is known, it *should* be supplied. Otherwise, it is computed using the functions `LieCharacteristic` and `LieType`.

If `Verify` is `false`, then assume G is isomorphic, possibly modulo scalars, to $(P)SL(3, q)$.

`SL3ElementToWord (G, g)`

If g is an element of G which has been constructively recognised to have central quotient isomorphic to $PSL(3, q)$, then return `true` and element of word group for G which evaluates to g, else `false`. This facilitates membership testing in G.

The constructive recognition algorithms for $SL(3, q)$ were developed by Lübeck, Magaard, and O'Brien [LMO07]. Its current implementation, which is part of the `CompositionTree` package, was developed by Bäärnhielm and O'Brien.

Example H65E11

We create $SL(3, 5^4)$ in its natural representation and recognise it. We then form its symmetric square and apply the recognition machinery to that.

```
> G := SL(3, 5^4);
> flag, phi, tau, gamma, delta := RecogniseSL3(G);
> w := PrimitiveElement (GF(5^4));
> g := GL(3, 5^4)! [1,2,1,0,w,1,0,0,w^-1];
> w := gamma (g);
> delta (w) eq g;
true
> G := ActionGroup(SymmetricSquare(GModule(G)));
> flag, phi, tau, gamma, delta := RecogniseSL3(G);
> phi;
Mapping from: GL(6, GF(5, 4)) to SL(3, GF(5, 4)) given by a rule [no inverse]
> g := Random(G);
> h := phi(g);
> h;
[$.1^40430   $.1^970]
[ $.1^5607 $.1^11606]
> k := tau(h);
> w := gamma(k);
> m := delta(w);
```

Recall that we are working modulo scalars. We conclude by testing whether a random element of $GL(6, 5^4)$ is contained in our group.

```
> IsScalar(m * g^-1);
true
> g := Random(GL(6, 5^4));
> SL3ElementToWord(G, g);
false
```

`RecogniseSL(G, d, q)`

`RecognizeSL(G, d, q)`

Use the Kantor-Seress algorithm to try to find an isomorphism between the finite group G (regarded as a black-box group) and $\mathrm{SL}(d, q)$ or $\mathrm{PSL}(d, q)$. The first return value indicates whether the attempt was successful. If so, then the second and third return values are mutually inverse homomorphisms (modulo scalars if $G \cong \mathrm{PSL}(d, q)$) from G to $\mathrm{SL}(d, q)$ and from $\mathrm{SL}(d, q)$ to G.
Warning: This function often returns `false` even when G is isomorphic to $\mathrm{SL}(d, q)$ or $\mathrm{PSL}(d, q)$, so it should be called repeatedly until it returns `true`!

65.3.6 Constructive Recognition of Symplectic Groups

`RecogniseSpOdd(G, d, q)`

`RecognizeSpOdd(G, d, q)`

Use the Kantor-Seress algorithm to try to find an isomorphism between the finite group G (regarded as a black-box group) and $\mathrm{Sp}(d, q)$ or $\mathrm{PSp}(d, q)$ for odd q. The first return value indicates whether the attempt was successful. If so, then the second and third return values are mutually inverse homomorphisms (modulo scalars if $G \cong \mathrm{PSp}(d, q)$) from G to $\mathrm{Sp}(d, q)$ and from $\mathrm{Sp}(d, q)$ to G.
Warning: This function often returns `false` even when G is isomorphic to $\mathrm{Sp}(d, q)$ or $\mathrm{PSp}(d, q)$, so it should be called repeatedly until it returns `true`!

`RecogniseSp4Even(G, q)`

`RecognizeSp4Even(G, q)`

Use an algorithm of Peter Brooksbank to try to find an isomorphism between the finite group G (regarded as a black-box group) and $\mathrm{Sp}(4, q)$ for even q. The first return value indicates whether the attempt was successful. If so, then the second and third return values are mutually inverse homomorphisms from G to $\mathrm{Sp}(d, q)$ and from $\mathrm{Sp}(d, q)$ to G. The third and fourth return values are mutually inverse homomorphisms from G to the word group W of G and from W to G.

65.3.7 Constructive Recognition of Unitary Groups

`RecogniseSU3(G, d, q)`

`RecognizeSU3(G, d, q)`

Use an algorithm of Peter Brooksbank to try to find an isomorphism between the finite group G (regarded as a black-box group) and $\mathrm{SU}(3, q)$ or $\mathrm{PSU}(3, q)$ for $q > 2$. The first return value indicates whether the attempt was successful. If so, then the second and third return values are mutually inverse homomorphisms (modulo scalars if $G \cong \mathrm{PSU}(3, q)$) from G to $\mathrm{SU}(3, q)$ and from $\mathrm{SU}(3, q)$ to G. The third and fourth return values are mutually inverse homomorphisms from G to the word group W of G and from W to G.

`RecogniseSU4(G, d, q)`

`RecognizeSU4(G, d, q)`

Use an algorithm of Peter Brooksbank to try to find an isomorphism between the finite group G (regarded as a black-box group) and $\mathrm{SU}(4,q)$ or $\mathrm{PSU}(4,q)$. The first return value indicates whether the attempt was successful. If so, then the second and third return values are mutually inverse homomorphisms (modulo scalars if $G \cong \mathrm{PSU}(4,q)$) from G to $\mathrm{SU}(4,q)$ and from $\mathrm{SU}(4,q)$ to G. The third and fourth return values are mutually inverse homomorphisms from G to the word group W of G and from W to G.

65.3.8 Constructive Recognition of $\mathrm{SL}(d,q)$ in Low Degree

Let $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$ with $q = p^f$, where V is the natural H-module. Let H act on an irreducible $\mathbf{F}_q$-module W of dimension at most d^2. Magaard, O'Brien & Seress [MOAS08] describe algorithms which, given as input the irreducible representation of H on W, construct a d-dimensional projective representation of H. Their implementations, prepared by Eamonn O'Brien, are described below.

`RecogniseSymmetricSquare (G)`

G is symmetric square representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$ and $d \geq 4$. Reconstruct H; if successful, then return `true` and H, otherwise `false`.

`SymmetricSquarePreimage (G, g)`

G is symmetric square representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$; return preimage of g in H.

`RecogniseAlternatingSquare (G)`

G is alternating square representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$ and $d \geq 3$. Reconstruct H; if successful, then return `true` and H, otherwise `false`.

`AlternatingSquarePreimage (G, g)`

G is alternating square representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$; return preimage of g in H.

`RecogniseAdjoint (G)`

G is adjoint representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$ and $d \geq 3$. Reconstruct H; if successful, then return `true` and H, otherwise `false`.

`AdjointPreimage (G, g)`

G is adjoint representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$; return preimage of g in H.

`RecogniseDelta (G)`

G is absolutely irreducible representation of $H \otimes H^{p^e}$, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$ and $d \geq 4$. Reconstruct H; if successful, then return `true` and H, otherwise `false`.

`DeltaPreimage (G, g)`

G is absolutely irreducible representation of $H \otimes H^{(}p^e)$, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$; return preimage of g in H.

Example H65E12

```
> G := SL(4, 3^2);
> G := SL(4, 9);
> M := GModule (G);
> M := SymmetricPower (M, 2);
> G := MatrixGroup (M);
> G := RandomConjugate (G);
> f, H := RecogniseSymmetricSquare (G);
> f;
true
> H;
MatrixGroup(4, GF(3^2))
Generators:
    [    0     1     0     0]
    [    0     0     0     1]
    [$.1^6     2     2   $.1]
    [    2   $.1     0   $.1]
    [    0     0     1     0]
    [$.1^2 $.1^7     1 $.1^6]
    [    1     2 $.1^6 $.1^6]
    [  $.1     0 $.1^7     0]
> g := Random (G);
>  h := SymmetricSquarePreimage (G, g);
> h;
[$.1^6     0     0 $.1^2]
[$.1^6     0 $.1^3     0]
[$.1^2 $.1^5     2   $.1]
[    0 $.1^3 $.1^5   $.1]
```

65.3.9 Constructive Recognition of Suzuki Groups

65.3.9.1 Introduction

A description of the functionality for constructive recognition and constructive membership testing of the Suzuki groups $\mathrm{Sz}(q)$, with $q = 2^{2m+1}$ for some $m > 0$ follows.

The main intrinsics of the package are `RecogniseSz(G)` which performs constructive recognition of $G \cong \mathrm{Sz}(q)$, `SzElementToWord(G, g)` which returns a `GrpSLPElt` for g in the generators of G, and `IsSuzukiGroup(G)` which is a non-constructive test for isomorphism between G and $\mathrm{Sz}(q)$.

Informative printing can be obtained using one of a number of verbose flags:

`SuzukiGeneral`, for the general routines.

`SuzukiStandard`, for the routines related to the standard copy.

`SuzukiConjugate`, for the routines related to conjugation.

`SuzukiTensor`, for the routines related to tensor decomposition.

`SuzukiMembership`, for the routines related to membership testing.

`SuzukiCrossChar`, for the routines related to cross-characteristic representations.

`SuzukiTrick`, for the routines related to the double coset trick.

`SuzukiNewTrick`, for the routines related to the stabiliser trick.

For each of the flags, the verbose level takes any value up to 10, with higher values resulting in more output.

65.3.9.2 Recognition Functions

`IsSuzukiGroup(G)`

Given a matrix group G, this function determines (non-constructively) whether or not G is isomorphic to $\mathrm{Sz}(q)$. The corresponding finite field cardinality q is also returned.

If the group G is defined over a field of odd characteristic or has degree greater than 4, the Monte Carlo algorithm of `LieType` is used. If G has degree 4 and is over a field of characteristic 2, then a fast Las Vegas algorithm is used, described in [Bää06a].

`RecogniseSz(G : parameters)`

`RecognizeSz(G : parameters)`

`Verify`	BoolElt	*Default :* `true`
`FieldSize`	RngIntElt	*Default :*
`Optimise`	BoolElt	*Default :* `false`

Let G be a group that is absolutely irreducible and is defined over a minimal field. This function constructively recognises G as a Suzuki group. If G is isomorphic to $\mathrm{Sz}(q)$, where q is the size of the defining field of G, then return:

Isomorphism from G to $\mathrm{Sz}(q)$.

Isomorphism from $\mathrm{Sz}(q)$ to G.

Map from G to the word group of G.

Map from the word group of G to G.

The isomorphisms are composed of maps that are defined by rules, so `Function` should be used on each component to avoid unnecessary built-in membership testing. The word group is the `GrpSLP` group which is the parent of the elements returned by `SzElementToWord`. In general this is not the same as `WordGroup(G)`, but is created from it using `AddRedundantGenerators`.

If `Verify` is true, then it is checked if G is isomorphic to $\mathrm{Sz}(q)$, using `IsSuzukiGroup`. In that case, `FieldSize` must be set to the correct value of q. Constructive recognition of 2.Sz(8) is also handled.

If `Optimise` is true, then the third map returns element in an optimised word group (using `AddRedundantGenerators`). Then each invocation of the map will be faster, but the initialisation will take longer.

The algorithms used for constructive recognition are described in [Bää06a] and [Bää05].

`SzElementToWord(G, g)`

If G has been constructively recognised as a Suzuki group, and if g is an element of G, then return `true` and a `GrpSLPElt` from the word group of G which evaluates to g, else return `false`.

This facilitates membership testing in G.

`SzPresentation(q)`

If $q = 2^{2m+1}$ for some $m > 0$, return a short presentation of $\mathrm{Sz}(q)$ on the Magma standard generators, i.e. the generators returned by the `Sz` intrinsic.

`SatisfiesSzPresentation(G)`

G is constructively recognised as $\mathrm{Sz}(q)$ for some q. Verify that it satisfies a presentation for this group.

`SuzukiIrreducibleRepresentation(F, twists :` *parameters*`)`

`CheckInput` BoolElt *Default :* `true`

Let F be a finite field of cardinality $q = 2^{2m+1}$ for some $m > 0$, and let *twists* be a sequence of n distinct integers in the range $[0 \dots 2m]$. The function returns an absolutely irreducible representation of $\mathrm{Sz}(q)$ having dimension 4^n, being a tensor product of twisted powers of the copy returned by the `Sz` intrinsic, where the twists are given by the input sequence.

If `CheckInput` is true, then it is verified that F and *twists* satisfy the above requirements. Otherwise this is not checked.

Example H65E13

We illustrate the basic facilities starting with a random conjugate of the standard version of the Suzuki group $Sz(32)$. We first perform non-constructive recognition.

```
> G := Sz(32);
> G ^:= Random(Generic(G));
> flag, q := SuzukiRecognition(G);
> flag, q eq 32;
true true
```

The next step is to perform constructive recognition. The explicit isomorphisms will be the values of `iso` and `inv`.

```
> flag, iso, inv, g2slp, slp2g := RecognizeSz(G);
> flag;
true
> iso, inv;
Mapping from: GrpMat: G to MatrixGroup(4, GF(2^5)) given by a rule [no inverse]
Mapping from: MatrixGroup(4, GF(2^5)) to GrpMat: G given by a rule [no inverse]
```

We now experiment with membership testing. We use Function to avoid MAGMA's built-in membership testing but in doing so, we may not obtain the shortest possible SLP.

```
> w := Function(g2slp)(G.1);
> #w;
284
```

The algorithm is probabilistic, so different executions will most likely give different results.

```
> ww := Function(g2slp)(G.1);
> w eq ww;
false
```

Note that the resulting SLPs are from a word group that is not the word group W corresponding to the defining generators of G. However, they can be coerced into W.

```
> W := WordGroup(G);
> NumberOfGenerators(Parent(w)), NumberOfGenerators(W);
7 3
> flag, ww := IsCoercible(W, w);
> flag;
true
> slp2g(w) eq Evaluate(ww, UserGenerators(G));
true
```

So there are two ways to get the element back. An alternative is to use the intrinsic `SzElementToWord`, which is better if the elements are not known to lie in the group.

```
> flag, ww := SzElementToWord(G, G.1);
> flag, slp2g(w) eq slp2g(ww);
```

```
true true
```

We take an element just outside the group.

```
> H := Sp(4, 32);
> flag, ww := SzElementToWord(G, H.1);
> flag;
false
> // in this case we will not get an SLP
> ww := Function(g2slp)(H.1);
> ww;
false
> SatisfiesSzPresentation(G);
true
```

Example H65E14

As a variation we apply the machinery to $2.Sz(8)$. We demonstrate constructive recognition and constructive membership testing.

```
> A := ATLASGroup("2Sz8");
> reps := MatRepKeys(A);
> G := MatrixGroup(reps[3]);
> Degree(G), CoefficientRing(G);
40 Finite field of size 7
> flag, iso, inv, g2slp, slp2g := RecognizeSz(G);
> flag;
true
> R := RandomProcess(G);
> g := Random(R);
> w := Function(g2slp)(g);
> slp2g(w) eq g;
true
```

Example H65E15

For the next example we consider a case where the dimension is large. We construct the Suzuki group in a 64-dimensional matrix representation and then take a random conjugate and also rewrite is over a smaller field.

```
> F := GF(2, 9);
> twists := [0, 3, 6];
> G := SuzukiIrreducibleRepresentation(F, twists);
> Degree(G), IsAbsolutelyIrreducible(G);
64 true
> G ^:= Random(Generic(G));
> flag, GG := IsOverSmallerField(G);
> flag, CoefficientRing(GG);
```

```
true Finite field of size 2^3
```

Non-constructive recognition is harder in this case and will give us the defining field size. Constructive recognition will decompose the tensor product.

```
> time SuzukiRecognition(GG);
true 512
Time: 2.330
> time flag, iso, inv, g2slp, slp2g := RecogniseSz(GG);
Time: 4.800
> iso;
Mapping from: GrpMat: GG to MatrixGroup(4, GF(2^9)) given by a rule [no inverse]
```

Constructive membership is again easy

```
> R := RandomProcess(GG);
> g := Random(R);
> time w := Function(g2slp)(g);
Time: 0.020
> // but SLP evaluation is harder in large dimensions
> time slp2g(w) eq g;
true
Time: 0.370
> time SatisfiesSzPresentation(GG);
true
Time: 10.930
```

Example H65E16

The final example will be in cross characteristic. We build a representation of $Sz(8)$ in cross characteristic.

```
> G := Sz(8);
> _, P := SuzukiPermutationRepresentation(G);
> // for example over GF(9)
> M := PermutationModule(P, GF(3, 2));
> factors := CompositionFactors(M);
> exists(m64){f : f in factors | Dimension(f) eq 64};
true
> m64;
GModule m64 of dimension 64 over GF(3^2)
> H := ActionGroup(m64);
> IsAbsolutelyIrreducible(H);
true
> flag, G := IsOverSmallerField(H);
> Degree(G), CoefficientRing(G);
64 Finite field of size 3
```

We actually end up with a group in characteristic 3.

```
> time flag, iso, inv, g2slp, slp2g := RecogniseSz(G);
```

```
Time: 3.490
> iso;
Mapping from: GrpMat: G to MatrixGroup(4, GF(2^3)) given by a rule [no inverse]
> R := RandomProcess(G);
> g := Random(R);
> time w := Function(g2slp)(g);
Time: 0.010
> time slp2g(w) eq g;
true
Time: 0.110
> time SatisfiesSzPresentation(G);
true
Time: 0.330
```

65.3.10 Constructive Recognition of Small Ree Groups

65.3.10.1 Introduction

This machinery provides functionality for constructive recognition and constructive membership testing of the small Ree groups ${}^2\mathrm{G}_2(q) = \mathrm{Ree}(q)$, with $q = 3^{2m+1}$ for some $m > 0$.

The important intrinsics are `RecogniseRee` which performs constructive recognition of $G \cong \mathrm{Ree}(q)$, `ReeElementToWord` which returns a `GrpSLPElt` for g in the generators of G, and `IsReeGroup` which is a non-constructive test for isomorphism between G and $\mathrm{Ree}(q)$.

There are a few verbose flags used in the package.

`ReeGeneral`, for the general routines.

`ReeStandard`, for the routines related to the standard copy.

`ReeConjugate`, for the routines related to conjugation.

`ReeTensor`, for the routines related to tensor decomposition.

`ReeMembership`, for the routines related to membership testing.

`ReeCrossChar`, for the routines related to cross-characteristic representations.

`ReeTrick`, for the routines related to the stabiliser trick.

`ReeInvolution`, for the routines related to involution centralisers.

`ReeSymSquare`, for the routines related to symmetric square decomposition.

All the flags can be set to values up to 10, with higher values resulting in more output.

65.3.10.2 Recognition Functions

`RecogniseRee(G : parameters)`

`RecognizeRee(G : parameters)`

`Verify`	BoolElt	*Default :* `true`
`FieldSize`	RngIntElt	*Default :*
`Optimise`	BoolElt	*Default :* `false`

G is absolutely irreducible and defined over minimal field. Constructively recognise G as a Ree group. If G is isomorphic to Ree(q) where q is the size of the defining field of G, then return:

Isomorphism from G to Ree(q).

Isomorphism from Ree(q) to G.

Map from G to the word group of G.

Map from the word group of G to G.

The isomorphisms are composed of maps that are defined by rules, so `Function` should be used on each component to avoid unnecessary built-in membership testing.

The word group is the `GrpSLP` which is the parent of the elements returned by `ReeElementToWord`. In general this is not the same as `WordGroup(G)`, but is created from it using `AddRedundantGenerators`.

If `Verify` is true, then it is checked that G is isomorphic to Ree(q), using `IsReeGroup`, otherwise this is not checked. In that case, `FieldSize` must be set to the correct value of q.

If `Optimise` is true, then the third map returns element in an optimised word group (using `AddRedundantGenerators`). Then each invocation of the map will be faster, but the initialisation will take longer.

The algorithms for constructive recognition are those of [Bää06b].

`ReeElementToWord(G, g)`

If G has been constructively recognised as a Ree group, and if g is an element of G, then return `true` and a `GrpSLPElt` from the word group of G which evaluates to g, else return `false`.

This facilitates membership testing in G.

`IsReeGroup(G)`

Determine (non-constructively) if G is isomorphic to Ree(q). The corresponding q is also returned.

If G is over a field of characteristic not 3 or has degree greater than 7, the Monte Carlo algorithm of `LieType` is used. If G has degree 7 and is over a field of characteristic 3, then a fast Las Vegas algorithm is used.

`ReeIrreducibleRepresentation(F, twists : parameters)`

`CheckInput` BoolElt *Default :* `true`

The finite field F must have size $q = 3^{2m+1}$ for some $m > 0$, and *twists* should be a sequence of n distinct pairs of integers (i, j) where i is 7 or 27 and j in the range $[0 \dots 2m]$.

Return an absolutely irreducible representation of $\mathrm{Ree}(q)$, a tensor product of twisted powers of the representation of dimension 7 or 27, where the twists are given by the input sequence.

If `CheckInput` is `true`, then it is verified that F and *twists* satisfy the above requirements. Otherwise this is not checked.

Example H65E17

Our first example shows off the recognition machinery for the Ree group defined over $\mathbf{F}_{27}$.

```
> SetSeed(1);
> F := GF(3, 3);
> G := ReeGroup(F);
> G ^:= Random(Generic(G));
> flag, q := ReeRecognition(G);
> flag, q eq #F;
true true
> flag, iso, inv, g2slp, slp2g := RecognizeRee(G);
> flag;
true
> iso, inv;
Mapping from: GrpMat: G to MatrixGroup(7, GF(3^3)) given by a rule [no inverse]
Mapping from: MatrixGroup(7, GF(3^3)) to GrpMat: G given by a rule [no inverse]
```

We now experiment with membership testing. As the algorithm is probabilistic, different executions will most likely give different results.

```
> w := Function(g2slp)(G.1);
> #w;
342
> ww := Function(g2slp)(G.1);
> w eq ww;
false
```

The resulting SLPs are from another word group but can be coerced into W.

```
> W := WordGroup(G);
> NumberOfGenerators(Parent(w)), NumberOfGenerators(W);
7 3
> flag, ww := IsCoercible(W, w);
> flag;
true
> // so there are two ways to get the element back
> slp2g(w) eq Evaluate(ww, UserGenerators(G));
```

```
true
```

If the elements are not known to lie in the group, a better alternative is to use the intrinsic `ReeElementToWord`. We take a generator of $\Omega(7, F)$ as an example of an element not lying in $G_2(27)$.

```
> flag, ww := ReeElementToWord(G, G.1);
> flag, slp2g(w) eq slp2g(ww);
true true
> H := Omega(7, #F);
> flag, ww := ReeElementToWord(G, H.1);
> flag;
false
> ww := Function(g2slp)(H.1);
> ww;
false
```

65.3.11 Constructive Recognition of Large Ree Groups

65.3.11.1 Introduction

This machinery provides functionality for constructive recognition and constructive membership testing of the large Ree groups ${}^2\mathrm{F}_4(q) = \mathrm{LargeRee}(q)$, with $q = 2^{2m+1}$ for some $m > 0$.

The important intrinsics are `RecogniseLargeRee` which performs constructive recognition of $G \cong \mathrm{LargeRee}(q)$, `LargeReeElementToWord` which returns a `GrpSLPElt` for g in the generators of G, and `IsLargeReeGroup` which is a non-constructive test for isomorphism between G and $\mathrm{LargeRee}(q)$.

There are a few verbose flags used in the package.

`LargeReeGeneral`, for the general routines.

`LargeReeStandard`, for the routines related to the standard copy.

`LargeReeConjugate`, for the routines related to conjugation.

`LargeReeRyba`, for the routines related to membership testing.

`LargeReeTrick`, for the routines related to the stabiliser trick.

`LargeReeInvolution`, for the routines related to involution centralisers.

All the flags can be set to values up to 10, with higher values resulting in more output.

65.3.11.2 Recognition Functions

`RecogniseLargeRee(G : parameters)`

`RecognizeLargeRee(G : parameters)`

`Verify`	BoolElt	*Default :* `true`
`FieldSize`	RngIntElt	*Default :*
`Optimise`	BoolElt	*Default :* `false`

G is absolutely irreducible and defined over minimal field. Constructively recognise G as a Large Ree group. If G is isomorphic to LargeRee(q) where q is the size of the defining field of G, then return:

Isomorphism from G to LargeRee(q).

Isomorphism from LargeRee(q) to G.

Map from G to the word group of G.

Map from the word group of G to G.

The isomorphisms are composed of maps that are defined by rules, so `Function` should be used on each component to avoid unnecessary built-in membership testing.

The word group is the `GrpSLP` which is the parent of the elements returned by `LargeReeElementToWord`. In general this is not the same as `WordGroup(G)`, but is created from it using `AddRedundantGenerators`.

If `Verify` is true, then it is checked that G is isomorphic to LargeRee(q), using `IsLargeRee`, otherwise this is not checked. In that case, `FieldSize` must be set to the correct value of q.

If `Optimise` is true, then the third map returns element in an optimised word group (using `AddRedundantGenerators`). Then each invocation of the map will be faster, but the initialisation will take longer.

`LargeReeElementToWord(G, g)`

If G has been constructively recognised as a Large Ree group, and if g is an element of G, then return `true` and a `GrpSLPElt` from the word group of G which evaluates to g, else return `false`.

This facilitates membership testing in G.

`IsLargeReeGroup(G)`

Determine (non-constructively) if G is isomorphic to LargeRee(q). The corresponding q is also returned.

If G is over a field of characteristic not 2 or has degree greater than 26, the Monte Carlo algorithm of `LieType` is used. If G has degree 26 and is over a field of characteristic 2, then a fast Las Vegas algorithm is used.

65.4 Properties of Finite Groups Of Lie Type

65.4.1 Maximal Subgroups of the Classical Groups

The `ClassicalMaximals` function, written by Derek Holt and Colva Roney-Dougal, returns a list of the maximal subgroups of the classical quasisimple groups in their natural representations, as returned by the MAGMA functions `SL, Sp, SU, Omega, OmegaPlus, OmegaMinus`. The list should be complete for dimensions up to 12 apart from a few omissions in $\Omega^+(8,q)$ which will be rectified in the near future.

There are also options to return the normalisers of these subgroups in various groups, such as $\mathrm{GL}(n,q)$, $\mathrm{GU}(n,q)$, that lie between the quasisimple group and its normaliser in the general linear group. These should be sufficient to enable the skilled user to determine the maximal subgroups of any group lying between the quasisimple groups and its normaliser.

According to the theorem of Aschbacher [Asc84] discussed earlier in this chapter, the maximal subgroups of a quasisimple classical group over a finite field lie in (at least) one of nine categories, which were listed in the Aschbacher Reduction section.

The subgroups in the first eight of these categories are said to be of *geometric type* and can be described in a uniform fashion. This description is the topic of the book [KL90]. They are returned in all dimensions by `ClassicalMaximals`. There is no such uniform description of the subgroups in the ninth class, which have to be classified separately in each dimension. The lists in the papers [HM01], [HM02] and [LÖ1] contain sufficient information in theory to compute these subgroups up to dimension 250, but currently this has been carried out only up to dimension 12.

`ClassicalMaximals(type, d, q : parameters)`

`classes`	SETENUM	*Default :* $\{1 \ldots 9\}$
`all`	BOOLELT	*Default :* `true`
`special`	BOOLELT	*Default :* `true`
`general`	BOOLELT	*Default :* `true`
`normaliser`	BOOLELT	*Default :* `true`
`novelties`	BOOLELT	*Default :* `false`

Return a list of representatives of the conjugacy classes of maximal subgroups of the quasisimple group of the specified type in dimension d over the field of order q. The string *type* must be one of `L, S, U, O, O+, O-`.

If the optional parameter `classes` is set to a proper subset of $\{1 \ldots 9\}$, then only the subgroups lying in the corresponding Aschbacher categories will be returned.

If the option `all` is set `false`, then representatives of the conjugacy classes under the action of the full automorphism group of the simple classical group will be returned: so this option will usually result in fewer subgroups in the returned list!

The option `special` only has effect for types `O, O+, O-`. When this is set to `true`, the normalisers of the subgroups in the appropriate group $\mathrm{SO}(d,q)$, $\mathrm{SO}^+(d,q)$ or $\mathrm{SO}^-(d,q)$ will be returned.

If the option `general` is set to `true`, then the normalisers of the subgroups in the appropriate group $\mathrm{GL}(d,q)$, $\mathrm{GU}(d,q)$, $\mathrm{GO}(d,q)$, $\mathrm{GO}^+(d,q)$ or $\mathrm{GO}^-(d,q)$ will be returned. (This option has not effect for type `S`.)

If the option `normaliser` is set to `true`, then the normalisers of the subgroups in the full normaliser of the quasisimple group in the general linear group (i.e. the group preserving the relevant form modulo scalars) will be returned. (For type `L` this has the same effect as setting *general* to `true`.)

If the option `novelties` is set `true`, then the intersections with the quasisimple group of any novelty maximal subgroups of any groups lying between the simple group and its full automorphism group will be returned. Use this option with caution, because the results are not guaranteed to be reliable!

65.4.2 Maximal Subgroups of the Exceptional Groups

Here follows some intrinsics for creating and conjugating maximal subgroups of Suzuki and Ree groups. The flags `SuzukiMaximals` and `ReeMaximals` may be used to produce verbose output.

`SuzukiMaximalSubgroups(G)`

If G has been constructively recognised as a Suzuki group, return a sequence of representatives of the maximal subgroups of G. Also returns sequences of `GrpSLPElt` of the generators of the subgroups, from the word group of G.

`SuzukiMaximalSubgroupsConjugacy(G, R, S)`

If G has been constructively recognised as a Suzuki group and if R and S are conjugate maximal subgroups of G, then return an element g of G that conjugates R to S. A `GrpSLPElt` from the word group of G, that evaluates to g, is also returned.

`ReeMaximalSubgroups(G)`

If G has been constructively recognised as a Ree group, return a sequence of representatives of the maximal subgroups of G. Also returns sequences of `GrpSLPElt` of the generators of the subgroups, from the word group of G.

`ReeMaximalSubgroupsConjugacy(G, R, S)`

If G has been constructively recognised as a Ree group and if R and S are conjugate maximal subgroups of G, then return an element g of G that conjugates R to S. A `GrpSLPElt` from the word group of G, that evaluates to g, is also returned. This is not implemented if R, S are Frobenius groups.

65.4.3 Sylow Subgroups of the Classical Groups

The MAGMA ClassicalSylow package written by Mark Stather provides functionality for constructing and conjugating the Sylow p-subgroups of the classical groups over finite fields in their natural representation, for any prime p. The classical groups may be created in MAGMA using the `GL, SL, Sp, GO, GOPlus, GOMinus, SO, SOPlus, SOMinus, Omega, OmegaPlus, OmegaMinus, GU, SU` intrinsics.

This package makes use of code to compute the classical form fixed by a group written by Derek Holt, and code to conjugate classical forms written by Colva Roney-Dougal.

The algorithms in this package are described in [Sta], which in turn makes use of the descriptions of the Sylow subgroups of the classical groups given in [Wei55], [CF64], [R. R57] and [Car72]. The conjugation algorithms make use of only the Meataxe, Smash, basic linear algebra and the solution of norm equations over finite fields.

`ClassicalSylow(G,p)`

The argument G must be a classical group in its natural representation, up to conjugation, with the exception of $GO(2m+1, 2^e)$. More precisely, it must be a conjugate of a group returned by one of the intrinsics `GL, SL, Sp, GO, GOPlus, GOMinus, SO, SOPlus, SOMinus, Omega, OmegaPlus, OmegaMinus, GU, SU`. p must be a prime number. The intrinsic returns a Sylow p-subgroup of G as a matrix group.

`ClassicalSylowConjugation(G,P,S)`

The argument G must be a classical group in its natural representation, up to conjugation, with the exception of $GO(2m+1, 2^e)$. More precisely, it must be a conjugate of a group returned by one of the intrinsics `GL, SL, Sp, GO, GOPlus, GOMinus, SO, SOPlus, SOMinus, Omega, OmegaPlus, OmegaMinus, GU, SU`. The groups P and S must be Sylow p-subgroups of G. The intrinsic returns an element $g \in G$ with $P^g = S$.

`ClassicalSylowNormaliser(G,P)`

In this case G must the full classical group in its natural representation, up to conjugation, with the exception of $GO(2m + 1, 2^e)$. More precisely, it must be a conjugate of a group returned by one of the intrinsics `GL, Sp, GO, GOPlus, GOMinus, GU`. The subgroup P must be a Sylow p-subgroup of G. The intrinsic returns the normaliser of P in G.

`ClassicalSylowToPC(G,P)`

The argument G must be a classical group in its natural representation, up to conjugation, with the exception of $GO(2m+1, 2^e)$. More precisely, it must be a conjugate of a group returned by one of the intrinsics `GL, SL, Sp, GO, GOPlus, GOMinus, SO, SOPlus, SOMinus, Omega, OmegaPlus, OmegaMinus, GU, SU`. The group P must be a Sylow p-subgroup of G. The intrinsic returns a PC group Q isomorphic to P, and also an isomorphism from P to Q and an isomorphism from Q to P.

Example H65E18

We construct a Sylow 7-subgroup P of $G = Sp(28, 17^2)$, take a random conjugate S of P and then find a conjugating element g that takes P to S.

```
> SetSeed(1);
> G := Sp(28,17^2);
> time P := ClassicalSylow(G,7);
Time: 0.080
> S := P^Random(G);
> time g := ClassicalSylowConjugation(G,P,S);
Time: 0.400
```

We next compute the normaliser of P in G.

```
> time N := ClassicalSylowNormaliser(G,P);
Time: 0.310
> // and a PC presentation of P
> time Pc, PtoPc, PctoP := ClassicalSylowToPC(G,P);
Time: 0.200
> Pc;
GrpPC : Pc of order 2401 = 7^4
PC-Relations:
    Pc.1^7 = Id(Pc),
    Pc.2^7 = Id(Pc),
    Pc.3^7 = Id(Pc),
    Pc.4^7 = Id(Pc)
> // We get inverse isomorphisms PtoPc and PctoP
> g := Random(P);
> PctoP(PtoPc(g)) eq g;
true
> x := Random(Pc);
> PtoPc(PctoP(x)) eq x;
true
```

65.4.4 Sylow Subgroups of Exceptional Groups

The flags `SuzukiSylow` and `ReeSylow` may be used to produce verbose output.

`SuzukiSylow(G, p)`

If G has been constructively recognised as a Suzuki group, and if p is a prime number, return a random Sylow p-subgroup S of G.

Also returns a list of `GrpSLPElt` from the word group of G, of the generators of S. If p does not divide $|G|$, then the trivial subgroup is returned.

`SuzukiSylowConjugacy(G, R, S, p)`

If G has been constructively recognised as a Suzuki group, if p is a prime number and if R and S are Sylow p-subgroups of G, then return an element g of G that conjugates R to S. A `GrpSLPElt` from the word group of G, that evaluates to g, is also returned.

Example H65E19

We demonstrate finding a conjugating element for Sylow subgroup in an example over a large field.

```
> q := 2^121;
> G := Sz(q);
> G ^:= Random(Generic(G));
> G := DerivedGroupMonteCarlo(G);
> NumberOfGenerators(G);
19
```

Non-constructive recognition is now a bit harder.

```
> time SuzukiRecognition(G);
true 26584559915698317458076141205606891 52
Time: 0.190
> time flag, iso, inv, g2slp, slp2g := RecogniseSz(G);
Time: 22.810
```

However, after this, each call to constructive membership testing is then easy.

```
> R := RandomProcess(G);
> g := Random(R);
> time w := Function(g2slp)(g);
Time: 0.060
> // evaluating SLPs always takes some time
> time slp2g(w) eq g;
true
Time: 1.250
```

We now create some Sylow subgroups and find conjugating elements.

```
> p := Random([x[1] : x in Factorization(q - 1)]);
> time R := SuzukiSylow(G, p);
Time: 1.370
> time S := SuzukiSylow(G, p);
Time: 1.310
> // that was easy, as is conjugating them
> time g, slp := SuzukiSylowConjugacy(G, R, S, p);
Time: 1.340
> slp2g(slp) eq g;
true
> #R, NumberOfGenerators(R);
23 1
```

```
> time R := SuzukiSylow(G, 2);
Time: 164.020
> time S := SuzukiSylow(G, 2);
Time: 171.740
> NumberOfGenerators(R), #R;
121 70673882591135373183331900029716740633099355875024758324864248051704791 04
> time g, slp := SuzukiSylowConjugacy(G, R, S, 2);
Time: 1.650
```

Creating the Sylow 2-subgroup is hard since they have so many generators. One the other hand, finding a conjugating element is relatively easy.

`ReeSylow(G, p)`

If G has been constructively recognised as a Ree group, and if p is a prime number, return a random Sylow p-subgroup S of G.

Also returns a list of `GrpSLPElt` from the word group of G, of the generators of S. If p does not divide $|G|$, then the trivial subgroup is returned.

`ReeSylowConjugacy(G, R, S, p)`

If G has been constructively recognised as $\mathrm{Ree}(q)$, if p is a prime number and if R and S are Sylow p-subgroups of G, then return an element g of G that conjugates R to S, and a `GrpSLPElt` from the word group of G, that evaluates to g, is also returned.

Currently, this is not implemented for odd p that divide $q^3 + 1$.

`LargeReeSylow(G, p)`

If G has been constructively recognised as a Large Ree group, and if p is a prime number, return a random Sylow p-subgroup S of G.

Also returns a list of `GrpSLPElt` from the word group of G, of the generators of S. If p does not divide $|G|$, then the trivial subgroup is returned.

Currently, this is not implemented for p that divide $q + 1$.

Example H65E20

Starting with the Ree group over the field $\mathbf{F}_{3^{3_1}}$, we construct Sylow p-subgroups for different primes p.

```
> m := 7;
> F := GF(3, 2 * m + 1);
> q := #F;
> q;
14348907
> G := ReeGroup(F);
> G ^:= Random(Generic(G));
> G := DerivedGroupMonteCarlo(G);
> NumberOfGenerators(G);
```

```
19
> ReeRecognition(G);
true 14348907
> flag, iso, inv, g2slp, slp2g := RecogniseRee(G);
> R := RandomProcess(G);
> g := Random(R);
> w := Function(g2slp)(g);
> slp2g(w) eq g;
true
```

We first create two Sylow p-subgroups of prime order and find a conjugating element. Note that 4561 divides the order of G exactly once and also divides $q - 1$.

```
> p := 4561;
> R := ReeSylow(G, p);
> S := ReeSylow(G, p);
> g, slp := ReeSylowConjugacy(G, R, S, p);
```

Thus $R^g = S$. In this case we also automatically get an SLP for the conjugating element.

```
> slp2g(slp) eq g;
true
> #R, NumberOfGenerators(R);
4561 1
```

Sylow 3-subgroups are harder: they have order 3^{45} and hence a considerable number of generators.

```
> time R := ReeSylow(G, 3);
Time: 3.730
> S := ReeSylow(G, 3);
> NumberOfGenerators(R), #R;
15 2954312706550833698643
> time g, slp := ReeSylowConjugacy(G, R, S, 3);
Time: 0.300
```

65.4.5 Conjugacy of Subgroups of the Classical Groups

`IsGLConjugate(H, K)`

Given H and K, both subgroups of the same general linear group $G = GL(n, q)$, return the value `true` if H and K are conjugate in G. The function returns a second value in the event that the subgroups are conjugate: an element z which conjugates H into K. The algorithm is described in Roney-Dougal [RD04].

65.4.6 Conjugacy of Elements of the Exceptional Groups

The flags `SuzukiElements` and `ReeElements` may be used to produce verbose output.

`SzConjugacyClasses(G)`

If G has been constructively recognised as a Suzuki group, return a list of conjugacy classes, using the same format as the `ConjugacyClasses` intrinsic.

`SzClassRepresentative(G, g)`

If G has been constructively recognised as a Suzuki group, and g is an element of G, return the conjugacy class representative h of g, such that h is in the list returned by `SzConjugacyClasses`. Also returns c in G such that $g^c = h$.

`SzIsConjugate(G, g, h)`

If G has been constructively recognised as a Suzuki group, and g and h are elements of G, determine if g is conjugate to h. If so, return `true` and an element c such that $g^c = h$, otherwise return `false`.

`SzClassMap(G)`

If G has been constructively recognised as a Suzuki group, return its class map, as in the `ClassMap` intrinsic.

`ReeConjugacyClasses(G)`

If G has been constructively recognised as a Ree group, return a list of conjugacy classes, using the same format as the `ConjugacyClasses` intrinsic.

65.4.7 Irreducible Subgroups of the General Linear Group

`IrreducibleSubgroups(n, q)`

Return the list of conjugacy classes of irreducible subgroups of $GL(n, q)$ where q is a prime power. At present, the dimension n is restricted to 2. The list is complete for characteristic at least 5. The algorithm is based on the classification of Flannery and O'Brien [FO05].

`IrreducibleSolubleSubgroups(n, q)`

Return the list of conjugacy classes of soluble irreducible subgroups of $GL(n, q)$ where q is a prime power. At present, the dimension n is restricted to 2 or 3. The list is complete for characteristic at least 5. The algorithm is based on the classification of Flannery and O'Brien [FO05].

Example H65E21

```
> L := IrreducibleSubgroups(2, 19^5);
> #[x : x in L | IsAbelian (x)];
552
```

```
> L := IrreducibleSolubleSubgroups(2, 97^2);
> #L;
10617
> L[7];
MatrixGroup(3, GF(97^2))
Generators:
    [$.1^8775 $.1^2037 $.1^6016]
    [$.1^6017 $.1^6705 $.1^7812]
    [$.1^7813 $.1^2817   $.1^33]
```

65.5 Atlas Data for the Sporadic Groups

Most of the functions described here use data derived from the Web Atlas. The data has been prepared for inclusion in MAGMA by Michael Downward and Eamonn O'Brien. It maintains Atlas names, conventions and orderings.

All of these functions, except `GoodBasePoints`, accept as input matrix or permutation groups. The algorithm underpinning `GoodBasePoints` due to O'Brien & Wilson [OW05].

`StandardGenerators(G, str : parameters)`

`Projective`	BOOLELT	*Default :* `false`
`AutomorphismGroup`	BOOLELT	*Default :* `false`

Construct standard generators for small quasisimple or sporadic group G having name *str*; words in SLP group defined on the defining generators of G are also obtained for the standard generators.

If G is sporadic and `AutomorphismGroup` is `true`, assume G is automorphism group of group having name *str*.

If standard generators found, return `true` and sequences of generators and corresponding words, else `false`.

Note: A return value of `false` only means that the algorithm's random search for standard generators did not succeed within the number of tries allowed. If the user is sure the group G matches the name *str*, then they should try the function again.

If G is absolutely irreducible matrix group and `Projective` is `true`, then construct standard generators possibly modulo centre of G.

This function currently works for all sporadic simple groups and all quasisimple groups for which the simple quotient has order at most 2×10^8. If you call it with an invalid value of *str*, then it will print out a list of all valid values.

`IsomorphismToStandardCopy(G, str : parameters)`

`Projective`	BoolElt	*Default :* `false`
`AutomorphismGroup`	BoolElt	*Default :* `false`

Use the StandardGenerators function to construct a (possibly projective) isomorphism from G to a standard copy of G. Options as for StandardGenerators. The first returned value indicates whether the call of StandardGenerators was successful.

`StandardPresentation(G, str : parameters)`

`Projective`	BoolElt	*Default :* `false`
`Generators`	SeqEnum	*Default :* []
`AutomorphismGroup`	BoolElt	*Default :* `false`

Return `true` if standard presentation is satisfied by generators of sporadic group G having name *str*, else `false`.

If `AutomorphismGroup` is `true`, assume G is automorphism group of sporadic group having name *str*.

Standard generators may be supplied as `Generators`, otherwise defining generators are assumed to be standard.

If G is absolutely irreducible matrix group and `Projective` is `true`, then verify presentation modulo centre of G.

`MaximalSubgroups(G, str : parameters)`

`Projective`	BoolElt	*Default :* `false`
`Generators`	SeqEnum	*Default :* []
`AutomorphismGroup`	BoolElt	*Default :* `false`

Construct **some** maximal subgroups for sporadic group G having name *str*. If `AutomorphismGroup` is `true`, assume G is automorphism group of sporadic group having name *str* and construct **some** of its maximal subgroups.

If standard generators supplied as `Generators` or found for G then return `true` and list of subgroups, else return `false`.

If G is absolutely irreducible matrix group and `Projective` is `true`, then construct standard generators and so subgroups possibly modulo centre of G.

`Subgroups(G, str : parameters)`

`Projective`	BoolElt	*Default :* `false`
`Generators`	SeqEnum	*Default :* []

Construct certain subgroups for sporadic group G having name *str*. If standard generators supplied as `Generators` or found for G then return `true` and list of subgroups, else return `false`.

If G is absolutely irreducible matrix group and `Projective` is `true`, then construct standard generators possibly modulo centre of G.

`GoodBasePoints(G, str : parameters)`

`Projective`	BoolElt	*Default :* `false`
`Generators`	SeqEnum	*Default :* []

If standard generators supplied as `Generators` or found for sporadic group G having name *str*, then return `true` and list of base points for G, else return `false`.

If G is absolutely irreducible and `Projective` is `true`, then standard generators are possibly modulo centre of G, and base points are correspondingly adjusted.

`SubgroupsData(str)`

Display stored subgroup data for sporadic group having name *str*.

`MaximalSubgroupsData (str : parameters)`

`AutomorphismGroup`	BoolElt	*Default :* `false`

Display stored data for some maximal subgroups of sporadic group having name *str*. If `AutomorphismGroup` is `true`, then display stored data for some maximal subgroups of automorphism group of sporadic group.

Example H65E22

The machinery is illustrated in the case of the sporadic Janko group J_1.

```
> G :=
> MatrixGroup<7, GF(11) |
> [ 9, 1, 1, 3, 1, 3, 3, 1, 1, 3, 1, 3, 3, 9, 1, 3, 1, 3, 3, 9, 1, 3, 1, 3,
>  3, 9, 1, 1, 1, 3, 3, 9, 1, 1, 3, 3, 3, 9, 1, 1, 3, 1, 3, 9, 1, 1, 3, 1, 3 ],
> [ 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0,
> 0, 1, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 10, 10, 0, 0, 0, 0, 0,0] >;
> flag, S := StandardGenerators (G, "J1");
> flag;
true
> StandardPresentation (G, "J1": Generators := S);
true
> flag, M:= MaximalSubgroups (G, "J1": Generators := S);
> #M;
7
> M[4];
rec<recformat<name: MonStgElt, parent: MonStgElt, generators: SeqEnum,
 group: Grp, order: RngIntElt, index: RngIntElt> |
     name := 19:6,
     parent := J1,
     group := MatrixGroup(7, GF(11))
     Generators:
         [ 0  1  4  3  3  4  7]
         [ 1  2  8  3  6  2  9]
         [ 4  8 10  1  6  0  9]
         [ 3  3  1  8  9  1 10]
```

```
    [ 3  6  6  9  1  3  7]
    [ 4  2  0  1  3  0  9]
    [ 7  9  9 10  7  9  0]
    [ 4  6  2  3  8  1  6]
    [ 8  1  3 10  2  7  4]
    [ 3  6  1  0  6  9  6]
    [ 2  3  6  9  0  3  7]
    [ 7  8  5  2  4  6  4]
    [10  4  5  2  8  6  8]
    [10  9  0  1  9  8  9],
order := 114,
index := 1540
>
```

65.6 Bibliography

[**Asc84**] M. Aschbacher. On the maximal subgroups of the finite classical groups. *Invent. Math*, 76:469–514, 1984.

[**Bää05**] H. Bäärnhielm. Tensor decomposition of the Suzuki groups. submitted, 2005.

[**Bää06a**] H. Bäärnhielm. Recognising the Suzuki groups in their natural representations. *J. Algebra*, 300(1):171–198, 2006.

[**Bää06b**] Henrik Bäärnhielm. Constructive recognition of the Ree groups. preprint, 2006.

[**BKPS02**] L. Babai, W. M. Kantor, P. P. Pálfy, and Á. Seress. Black-box recognition of finite simple groups of Lie type by statistics of element orders. *J. Group Theory*, 5:383–401, 2002.

[**BLGN$^+$03**] R. Beals, C. R. Leedham-Green, A. C. Niemeyer, C. E. Praeger, and A. Seress. A black-box algorithm for recognising finite symmetric and alternating groups, I. *Trans. Amer. Math. Soc.*, 2003. To appear.

[**BP00**] Sergey Bratus and Igor Pak. Fast constructive recognition of a black box group isomorphic to S_n or A_n using Goldbach's conjecture. *J. Symbolic Comp.*, 29:33–57, 2000.

[**Car72**] R. Carter. *Simple Groups of Lie Type.* John Wiley & Sons, London, New York, Sydney, Toronto, 1972.

[**CF64**] R. Carter and P. Fong. The Sylow 2-subgroups of the finite classical groups. *Journal of Algebra*, 1:139–151, 1964.

[**CLG97a**] Frank Celler and Charles R. Leedham-Green. Calculating the Order of an Invertible Matrix. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 55–60. AMS, 1997.

[**CLG97b**] Frank Celler and C.R. Leedham-Green. A non-constructive recognition algorithm for the special linear and other classical groups. In *Groups and computation II (New Brunswick, NJ, 1995)*, volume 28 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 61–67. Amer. Math. Soc., 1997.

[**CLGM$^+$95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

[**CLGO06**] M.D.E. Conder, C.R. Leedham-Green, and E.A. O'Brien. Constructive recognition for $PSL(2, q)$. *Trans. Amer. Math. Soc.*, 358:1203–1221, 2006.

[**FO05**] D.L. Flannery and E.A. O'Brien. Linear groups of small degree over finite fields. *Internat. J. Algebra and Comput.*, 15:467–502, 2005.

[**HM01**] G. Hiß and G. Malle. Low-dimensional representations of quasi-simple groups. *LMS J. Comput. Math.*, 4:22–63, 2001.

[**HM02**] G. Hiß and G. Malle. Corrigenda: Low-dimensional representations of quasi-simple groups. *LMS J. Comput. Math.*, 5:95–126, 2002.

[**HRT01**] R. B. Howlett, L. J. Rylands, and D. E. Taylor. Matrix generators for exceptional groups of Lie type. *J. Symbolic Comput.*, 31(4):429–445, 2001.

[**KL90**] Peter Kleidman and Martin Liebeck. *The Subgroup Structure of the Finite Classical Groups*, volume 129 of *London Math. Soc. Lecture Note Ser.* CUP, Cambridge, 1990.

[**LÖ1**] F. Lübeck. Small degree representations of finite Chevalley groups in defining ch aracteristic. *LMS J. Comput. Math.*, 4:135–169, 2001.

[**LMO07**] F Lübeck, K Magaard, and E.A. O'Brien. Constructive recognition of $\mathrm{SL}_3(q)$. *J. Algebra*, 2007:617–633, 2007.

[**LO07**] Martin Liebeck and E.A. O'Brien. Finding the characteristic of a group of Lie type. *J. London Math. Soc.*, 2007.

[**MOAS08**] Kay Magaard, E. A. O'Brien, and Ákos Seress. Recognition of small dimensional representations of general linear groups. *J. Austral. Math. Soc.*, 85: 229–250, 2008.

[**NP92**] Peter M. Neumann and Cheryl E. Praeger. A Recognition Algorithm for Classical Groups. *Proc. London Math. Soc.*, 65(3):555–603, 1992.

[**NP97**] Alice C. Niemeyer and Cheryl E. Praeger. Implementing a Recognition Algorithm for Classical Groups. In *Groups and computation II (New Brunswick, NJ, 1995)*, volume 28 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 273–296. Amer. Math. Soc., 1997.

[**NP98**] Alice C. Niemeyer and Cheryl E. Praeger. A Recognition Algorithm for Classical Groups over Finite Fields. *Proc. London Math. Soc.*, 77(3):117–169, 1998.

[**NP99**] Alice C. Niemeyer and Cheryl E. Praeger. A Recognition Algorithm for Non-Generic Classical Groups over Finite Fields. *J. Austral. Math. Soc. Ser. A*, 67:223–253, 1999.

[**OW05**] E.A. O'Brien and R.A. Wilson. Subgroup chains in matrix groups. *preprint*, 2005.

[**Pra99**] Cheryl E. Praeger. Primitive prime divisor elements in finite classical groups. In *Proc. of Groups St. Andrews 1997 in Bath II*, number 261 in London Math. Soc. Lecture Notes Series, pages 605–623. Cambridge Univ. Press, 1999.

[**RD04**] Colva M. Roney-Dougal. Conjugacy of subgroups of the general linear group. *Experiment. Math.*, 13:151–163, 2004.

[**R.R57**] R.Ree. On some simple groups defined by Chevalley. *Trans. Am. Math. Soc.*, 84:392–400, 1957.

[**RT98**] L.J. Rylands and D.E. Taylor. Matrix generators for the orthogonal groups. *J. Symbolic Comp.*, 25:351–360, 1998.

[**Sta**] M. Stather. Constructive Sylow Theorems for the Classical Groups. to appear in Journal of Algebra.

[**Tay87**] Don Taylor. Pairs of Generators for Matrix Groups. I. *Cayley Bulletin 3*, 1987.

[**Wei55**] A. Weir. Sylow p-subgroups of the classical groups over finite fields with characteristic prime to p. *Proc. Am. Math. Soc*, 6:529–533, 1955.

66 DATABASES OF GROUPS

Chapter 66
DATABASES OF GROUPS

66.1 Introduction

This chapter describes the use of the various databases of groups that form part of MAGMA. The available databases are as follows:

Small Groups: This database is constructed by Hans Ulrich Besche, Bettina Eick and Eamonn O'Brien [BE99a, BEO01, BE99b, O'B90, BE01, O'B91, MNVL04, OVL05, DE05], contains the following groups:

- All groups of order up to 2000, excluding the groups of order 1024.
- The groups whose order is the product of at most 3 primes.
- The groups of order dividing p^6 for p a prime.
- The groups of order $q^n p$, where q^n is a prime-power dividing 2^8, 3^6, 5^5 or 7^4 and p is a prime different to q.
- The groups of square-free order. For a different mechanism for accessing the p-groups in this collection, see Section 66.3, specifically the functions `SearchPGroups` and `CountPGroups`. These functions also access groups of order p^7.

p-groups: MAGMA contains the means to construct all p-groups of order p^n where $n \leq 7$. The data used in the constructions was supplied by Hans Ulrich Besche, Bettina Eick, Eamonn O'Brien, Mike Newman and Michael Vaughan-Lee [BE99a, BEO01, BE99b, O'B90, BE01, O'B91, MNVL04, OVL05].

Metacyclic p-groups: MAGMA is able to construct all metacyclic groups of order p^n. This machinery was developed by Mike Newman, Eamonn O'Brien, and Michael Vaughan-Lee.

Perfect Groups: This database contains all perfect groups up to order 50000, and many classes of perfect groups up to order one million. Each group is defined by means of a finite presentation. Further information is also provided which allows the construction of permutation representations. This database was constructed by Derek Holt and Willem Plesken [HP89].

Almost Simple Groups: This database contains information about every group G, where $S \leq G \leq \mathrm{Aut}(S)$ and S is a simple group of order less than 16000000, or S is one of M_{24}, HS, J_3, McL, $Sz(32)$ or $L_6(2)$.

Transitive Permutation Groups: This database is a MAGMA version of the database of transitive permutation groups constructed by Alexander Hulpke [Hul05] (for degree up to 30) and Cannon and Holt [CH08]. It contains all transitive permutation groups having degree up to 32.

Primitive Permutation Groups: This is a database containing all primitive permutation groups having degree less than 4095 as determined by Sims (for degree ≤ 50), Roney-Dougal and Unger [RDU03] (for degree < 1000), Roney-Dougal [RD05] (for degree < 2500), and Coutts, Quick and Roney-Dougal [CQRD11] (for degree < 4096).

Rational Maximal Matrix Groups: This contains the rational maximal finite matrix groups and their invariant forms, for small dimensions (up to 31) as determined by Gabi Nebe and Willem Plesken [NP95, Neb96]. Each entry can be accessed either as a matrix group or as a lattice.

Quaternionic Matrix Groups: A database of the finite absolutely irreducible subgroups of $\mathrm{GL}_n(\mathcal{D})$ where $\mathcal{D}$ is a definite quaternion algebra whose centre has degree d over $\mathbf{Q}$ and $nd \leq 10$. Each entry can be accessed either as a matrix group or as a lattice. The database was constructed by Gabi Nebe [Neb98].

Irreducible Matrix Groups: A database of the irreducible subgroups of $\mathrm{GL}_n(p)$, p prime, $n \geq 1$ and $p^n < 2500$. The groups were determined by Colva Roney-Dougal and William Unger [RDU03] (for $p^n < 1000$) and Roney-Dougal [RD05].

Soluble Irreducible Groups: This database contains one representative of each conjugacy class of irreducible soluble subgroups of $\mathrm{GL}(n, p)$, p prime, for $n > 1$ and $p^n < 256$. It was constructed by Mark Short [Sho92].

ATLAS Groups: This database contains representations of nearly simple groups, as in the Birmingham ATLAS of Finite Group Representations. The data was supplied by Rob Wilson.

66.2 Database of Small Groups

Magma includes the Small Groups Library prepared by Besche, Eick and O'Brien. For a description of the algorithms used to generate these groups, details on the data structures used and applications we refer to [BE99a, BEO01, BE99b, O'B90, BE01, O'B91, MNVL04] and the references therein.

The Small Groups Library contains the following groups.

- All groups of order up to 2000, excluding the groups of order 1024.
- The groups whose order is a product of at most 3 primes.
- The groups of order dividing p^6 for p a prime.
- The groups of order $q^n p$, where q^n is a prime-power dividing 2^8, 3^6, 5^5 or 7^4 and p is a prime different to q.

The descriptions of the groups of order p^4, p^5, p^6 for $p > 3$ were contributed by Boris Girnat, Robert McKibbin, M.F. Newman, E.A. O'Brien, and M.R. Vaughan-Lee.

The Magma version of this library uses the same internal data format as the implementation available in GAP. In particular, the numbering of the groups of a given order in both packages is the same.

For a different mechanism for accessing the p-groups in this collection, see the `66.3` section, specifically the functions `SearchPGroups` and `CountPGroups`. These functions also access the groups of order p^7 (contributed by O'Brien and Vaughan-Lee).

66.2.1 Basic Small Group Functions

Many of the functions in this section have an optional parameter `Search`. It can be used to limit the small group search to soluble (`Search := "Soluble"`) or insoluble (`Search := "Insoluble"`) groups. The default is `Search := "All"`, which allows all groups to be considered.

When a group is extracted from the database, it is returned as a `GrpPC` if it is soluble, or as a `GrpPerm` if it is insoluble.

When using the small groups database for an extended search, it is advisable to open the database using the function `SmallGroupDatabase`, which opens the database and returns a reference to it. This reference can then be passed as first argument to most of the functions described below, and will save that function from opening and closing the database for itself. Doing so will reduce the number of file operations when a lot of use is made of the database. When the database is no longer needed, it can be closed using the `delete` statement.

```
SmallGroupDatabase()
```

```
OpenSmallGroupDatabase()
```

Open the small groups database (for extended search) and return a reference to it. This reference may be passed to other functions so that they do fewer file operations.

```
delete D
```

Close the small groups database `D` and free the resources associated with its use.

```
SmallGroupDatabaseLimit()
```

```
SmallGroupDatabaseLimit(D)
```

The limiting order up to which all groups (except those of order 1024) are stored in the database of small groups, that is, currently 2000.

```
IsInSmallGroupDatabase(o)
```

```
IsInSmallGroupDatabase(D, o)
```

Return `true` if the groups of order o are contained in the database and `false` otherwise. This function can be used to check whether o is a legitimate argument for other functions described in this section, avoiding runtime errors in user written loops or functions.

```
NumberOfSmallGroups(o)
```

```
NumberOfSmallGroups(D, o)
```

Given a positive integer o, return the number of groups of order o in the database. If the groups of order o are not contained in the database, 0 is returned. This function can be used to check whether a pair o, n defines a group contained in the small groups database, that is, whether it is a legitimate argument for other functions described in this section, avoiding runtime errors in user written loops or functions.

`SmallGroup(o, n)`

`SmallGroup(D, o, n)`

`Group(D, o, n)`

Given a positive integer o, such that the groups of order o are contained in the small groups library, and a positive integer n, return the n-th group of order o in the database. If the groups of order o are not contained in the database or if n exceeds the number of groups of order o in the database, an error is reported. The function `NumberOfSmallGroups` can be used to check whether the arguments are valid.

`SmallGroup(o:` *parameters*`)`

`SmallGroup(D, o:` *parameters*`)`

`Search` MonStgElt *Default : "All"*

Given a positive integer o, such that the groups of order o are contained in the small groups library, return the first group of order o in the database meeting the search criterion set by the parameter `Search`. If the groups of order o are not contained in the database, an error is reported. The function `IsInSmallGroupDatabase` can be used to check whether o is a valid argument for this function.

`SmallGroup(o, f:` *parameters*`)`

`SmallGroup(D, o, f:` *parameters*`)`

`Search` MonStgElt *Default : "All"*

Given a positive integer o such that the groups of order o are contained in the small groups library and a predicate f (as a function or intrinsic), return the first group of order o in the database meeting the search criterion set by the parameter `Search`, which satisfies f.

`IsSoluble(D, o, n)`

`IsSolvable(D, o, n)`

`SmallGroupIsSoluble(o, n)`

`SmallGroupIsSoluble(D, o, n)`

`SmallGroupIsSolvable(o, n)`

`SmallGroupIsSolvable(D, o, n)`

Return `true` iff `SmallGroup(o, n)` is soluble. This function does not load the group. If the group specified by the arguments does not exist in the database, an error is reported. The function `NumberOfSmallGroups` can be used to check whether the arguments are valid.

`SmallGroupIsInsoluble(o, n)`

`SmallGroupIsInsoluble(D, o, n)`

`SmallGroupIsInsolvable(o, n)`

`SmallGroupIsInsolvable(D, o, n)`

Return `true` iff `SmallGroup(o, n)` is insoluble. This function does not load the group. If the group specified by the arguments does not exist in the database, an error is reported. The function `NumberOfSmallGroups` can be used to check whether the arguments are valid.

`SmallGroup(o, f:` *parameters*`)`

`SmallGroup(S, f:` *parameters*`)`

`SmallGroup(D, o, f:` *parameters*`)`

`SmallGroup(D, S, f:` *parameters*`)`

`Search` MonStgElt *Default :* "*All*"

Given a sequence S of orders or a single order o contained in the database and a predicate f (as a function or intrinsic), return the first group with order in S or equal to o, respectively, which meets the search criterion set by the parameter `Search` and satisfies f.

`SmallGroups(o:` *parameters*`)`

`SmallGroups(D, o:` *parameters*`)`

`Search` MonStgElt *Default :* "*All*"

`Warning` BoolElt *Default :* `true`

Given an order o contained in the database, return a list of all groups of order o, meeting the search criterion set by the parameter `Search`. Some orders will produce a very large sequence of groups – in such cases a warning will be printed unless the user specifies `Warning := false`.

`SmallGroups(S:` *parameters*`)`

`SmallGroups(D, S:` *parameters*`)`

`Search` MonStgElt *Default :* "*All*"

`Warning` BoolElt *Default :* `true`

Given a sequence S of orders contained in the database, return a list of all groups with order in S, meeting the search criterion set by the parameter `Search`. The resulting sequence may be very long – in such cases a warning will be printed unless the user specifies `Warning := false`.

`SmallGroups(o, f: parameters)`

`SmallGroups(D, o, f: parameters)`

`Search` MonStgElt *Default : "All"*

Given an order o contained in the database and a predicate (function or intrinsic) f, return a list containing all groups G of order o, meeting the search criterion set by the parameter `Search` and satisfying `f(G) eq true`.

`SmallGroups(S, f: parameters)`

`SmallGroups(D, S, f: parameters)`

`Search` MonStgElt *Default : "All"*

Given a sequence S of orders contained in the database and a predicate (function or intrinsic) f, return a list containing all groups G with order in S, meeting the search criterion set by the parameter `Search` and satisfying `f(G) eq true`.

Example H66E1

(1) We find the non-abelian groups of order 27.

```
> list := SmallGroups(27, func<x|not IsAbelian(x)> );
> list;
[*
GrpPC of order 27 = 3^3
PC-Relations:
  $.2^$.1 = $.2 * $.3,

GrpPC of order 27 = 3^3
PC-Relations:
  $.1^3 = $.3,
  $.2^$.1 = $.2 * $.3
*]
```

(2) We get the first group in the database with derived length greater than 2.

```
> G := SmallGroup([1..100], func<x|DerivedLength(x) gt 2>);
> G;
GrpPC of order 24 = 2^3 * 3
PC-Relations:
  G.1^3 = Id(G),
  G.2^2 = G.4,
  G.3^2 = G.4,
  G.4^2 = Id(G),
  G.2^G.1 = G.3,
  G.3^G.1 = G.2 * G.3,
```

```
  G.3^G.2 = G.3 * G.4
```

(3) Now for a list of the insoluble groups of order 240. The insoluble groups in the database are returned as permutation groups.

```
> list := SmallGroups(240:Search:="Insoluble");
> #list;
8
> list[7];
Permutation group acting on a set of cardinality 7
  (1, 2, 3, 4)
  (1, 5, 2, 4, 3)(6, 7)
```

(4) The groups of order $2432 = 2^7 \cdot 19$ should be contained in the small groups database. We check this using the function `IsInSmallGroupDatabase`...

```
> IsInSmallGroupDatabase(2432);
true
```

... and determine the number of groups of order 2432.

```
> NumberOfSmallGroups(2432);
19324
```

(5) We find all groups of order 7^6 with cyclic centre of order 7^2.

```
> f := function (G)
>   Z := Centre (G);
>   return IsCyclic (Z) and #Z eq 7^2;
> end function;
> P := SmallGroups(7^6, f);
> #P;
30
> NumberOfSmallGroups(7^6);
860
```

66.2.2 Processes

A small group process enables iteration over all groups of specified orders satisfying a given predicate, without having to create and store all such groups together.

A small group process is created via the function `SmallGroupProcess` (in various forms). The standard process functions `IsEmpty`, `Current`, `CurrentLabel` and `Advance` can then be applied to the process.

The functions used to create a small group process all have a parameter `Search` attached to them. It can be used to limit the small group search to soluble (`Search := "Soluble"`) or insoluble (`Search := "Insoluble"`) groups. The default is `Search := "All"`, which allows all groups to be considered.

The `Process` functions described below do not have a variant with the database as first argument, as each process opens the database for an extended search automatically.

`SmallGroupProcess(o: parameters)`

`Search` MonStgElt *Default :* "*All*"

Given an order o contained in the small groups database, return a small group process which will iterate though all groups of order o meeting the search criterion set by the parameter `Search`.

`SmallGroupProcess(S: parameters)`

`Search` MonStgElt *Default :* "*All*"

Given a sequence S of orders contained in the small groups database, return a small group process which will iterate though all groups with order in the sequence S meeting the search criterion set by the parameter `Search`.

`SmallGroupProcess(o, f: parameters)`

`Search` MonStgElt *Default :* "*All*"

Given an order o contained in the small groups database and a predicate f (as function or intrinsic), return a small group process which will iterate though all groups of order o, which meet the search criterion set by the parameter `Search` and satisfy the predicate f.

`SmallGroupProcess(S, f: parameters)`

`Search` MonStgElt *Default :* "*All*"

Given a sequence S of orders contained in the small groups database and a predicate f (as function or intrinsic), return a small group process which will iterate though all groups with order in the sequence S, which meet the search criterion set by the parameter `Search` and satisfy the predicate f.

`IsEmpty(p)`

Returns `true` if the process p has passed its last group.

`Current(p)`

Return the current group of the process p.

`CurrentLabel(p)`

Return the label of the current group of the process p. That is, return o and n such that the current group is `SmallGroup(o, n)`.

`Advance(~p)`

Move the process p to its next group.

Example H66E2

We use a small group process to look at all the groups of order 128. We find the nilpotency class of each of them.

```
> P := SmallGroupProcess(128);
> count := {* *};
> repeat
>   G := Current(P);
>   Include(~count, NilpotencyClass(G));
>   Advance(~P);
> until IsEmpty(P);
> count;
{* 1^^15, 2^^947, 3^^1137, 4^^197, 5^^29, 6^^3 *}
```

66.2.3 Small Group Identification

The following functions perform the inverse operation to the small group functions described earlier. Given a group G such that a group isomorphic to G is in the database and identification of groups of order $|G|$ is supported, the identification functions return a pair $\langle o, n \rangle$ so that `SmallGroup(o, n)` is isomorphic to G.

Note that identifying a finitely presented group involves the construction of a permutation representation of this group, which may fail. We refer to the description of `IdentifyGroup` in Chapter 70 for details.

`IdentifyGroup(G)`

Locate the pair of integers $\langle o, n \rangle$ so that `SmallGroup(o, n)` is isomorphic to G. If there is no such group in the database or if identification of groups of order $|G|$ is not supported, then an error will result. The function `CanIdentifyGroup` can be used to test whether groups of a certain order can be identified; this may be useful for avoiding runtime errors in user written loops or functions.

`CanIdentifyGroup(o)`

Return `true` if identification of groups of order o in the database is supported. This function can be used to check whether a group is a legitimate argument for the functions `IdentifyGroup` described above, avoiding runtime errors in user written loops or functions.

Example H66E3

We identify a permutation group in the small group database, and get an isomorphic group from the database.

```
> G := DihedralGroup(10);
> G;
Permutation group G acting on a set of cardinality 10
Order = 20 = 2^2 * 5
  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  (1, 10)(2, 9)(3, 8)(4, 7)(5, 6)
> IdentifyGroup(G);
<20, 4>
> H := SmallGroup(20, 4);
> H;
GrpPC : H of order 20 = 2^2 * 5
PC-Relations:
    H.1^2 = Id(H),
    H.2^2 = Id(H),
    H.3^5 = Id(H),
    H.3^H.1 = H.3^4
```

66.2.4 Accessing Internal Data

The following functions provide access to data used internally by the Small Groups Library for representing groups. They are included just for completeness and are intended to be used by experts only. In particular, we do not give a detailed explanation of the (complicated) data format in this manual.

`Data(D, o, n)`

Returns the data from which the group number n of order o in D is constructed. The format and the meaning of the items in the returned list depends on the group indicated by the pair (o, n).

`SmallGroupEncoding(G)`

Given a finite solvable group G in the category `GrpPC`, return two integers c and o encoding the power conjugate presentation of G.

The second return value is the order of G. The first return value is an integer specifying the power conjugate relations in the presentation of G.

`SmallGroupDecoding(c, o)`

Given two integers c and o encoding a power conjugate presentation G, return G as a group in the category `GrpPC`.

The second argument is the order of G. The first return value is an integer specifying the power conjugate relations in the presentation G.

Example H66E4

(1) We extract a power conjugate presentation from the database and compute its encoding.

```
> D := SmallGroupDatabase();
> G := SmallGroup(D,1053,51);
> Category(G);
GrpPC
>  SmallGroupEncoding(G);
1002867124873971659396781 73 1053
```

(2) The second group of order 525 in the small groups database is stored as encoded power conjugate presentation.

```
> Data(D,525,2);
[* code, 666501 *]
```

We can create the corresponding group by decoding this information.

```
> G := SmallGroupDecoding(666501, 525);
> G;
GrpPC : G of order 525 = 3 * 5^2 * 7
PC-Relations:
    G.1^3 = Id(G),
    G.2^5 = G.4,
    G.3^7 = Id(G),
    G.4^5 = Id(G)
```

This gives the same presentation as accessing the database in the "usual" way.

```
> SmallGroup(D,525,2);
GrpPC of order 525 = 3 * 5^2 * 7
PC-Relations:
    $.1^3 = Id($),
    $.2^5 = $.4,
    $.3^7 = Id($),
    $.4^5 = Id($)
```

66.3 The p-groups of Order Dividing p^7

Magma contains the means to construct all p-groups of order p^n where $n \leq 7$. This section describes the functions for accessing these constructions. The data used in the constructions was supplied by Hans Ulrich Besche, Bettina Eick, Eamonn O'Brien, Mike Newman and Michael Vaughan-Lee [BE99a, BEO01, BE99b, O'B90, BE01, O'B91, MNVL04, OVL05].

`SearchPGroups(p, n: parameters)`

Produce a sequence of groups of order p^n satisfying the conditions specified by the following parameters. The restrictions on the order are $n \leq 7$ or $p = 2$ and $n \leq 9$.

`Rank` SetEnum *Default :* $\{1, \ldots n\}$

All groups returned will have Frattini quotient rank in `Rank`. This parameter may also be set to a single integer.

`Class` SetEnum *Default :* $\{1, \ldots n\}$

All groups returned will have p-class in `Class`. This parameter may also be set to a single integer.

`Select` Program *Default :* `true`

The parameter must be set to a program returning either `true` or `false` when given a p-group satisfying the above conditions. All groups G returned will then satisfy `Select(G) eq true`.

`Limit` RngIntElt *Default :* 0

If `Limit` is set to a positive number n, then the program may end its search and return when there are at least n groups found.

`CountPGroups(p, n: parameters)`

Count the number of groups of order p^n satisfying the conditions specified by the parameters. The parameters are the same as for `SearchPGroups`, except that the `Limit` parameter is ignored.

Example H66E5

We search the groups of order 19^7 for specific examples. There are, in total, 9380741 groups with this order. We start with a search for those of rank 5, class 3, and exponent 19. Since we do not set the `Limit` parameter, we will get a sequence containing all the examples.

```
> time Q := SearchPGroups(19, 7:Rank := 5, Class := 3,
>    Select := func<G|IsPrime(Exponent(G))> );
Time: 0.050
> #Q;
4
> Q[1];
GrpPC of order 893871739 = 19^7
PC-Relations:
  $.2^$.1 = $.2 * $.6,
```

```
  $.6^$.1 = $.6 * $.7
```

This time we limit the number returned.

```
> time    Q := SearchPGroups(19, 7:Rank := 4, Class := {3,4},
>    Select := func<G|IsPrime(Exponent(G))>, Limit := 5);
Time: 13.090
> #Q;
5
> [pClass(G):G in Q];
[ 3, 3, 3, 3, 3 ]
> time    Q4 := SearchPGroups(19, 7:Rank := 4, Class := 4,
>   Select := func<G|IsPrime(Exponent(G))>, Limit := 5);
Time: 0.150
> #Q4;
6
```

Note that the limit is not always adhered to exactly. We can also count the number of groups with our property.

```
> time CountPGroups(19, 7:Rank := 4, Class := {3,4},
>   Select := func<G|IsPrime(Exponent(G))>);
Time: 334.720
43
> time CountPGroups(19, 7:Rank := 4, Class := 4,
>    Select := func<G|IsPrime(Exponent(G))>);
10
Time: 0.310
```

66.4 Metacyclic p-groups

Magma contains functions for constructing all metacyclic groups of order p^n. It can also decide if a given p-group is metacyclic, construct invariants which distinguish this metacyclic group from all others of this order, and construct a standard presentation for the group.

This section describes the functions for accessing these algorithms. The functions were developed by Mike Newman, Eamonn O'Brien, and Michael Vaughan-Lee.

`MetacyclicPGroups(p, n: parameters)`

Return a list of the metacyclic groups of order p^m, where p is a prime and n is a positive integer.

`PCGroups` BoolElt *Default :* `true`

If `true`, the groups returned are in category GrpPC, otherwise they are in category GrpFP – this will be faster if the groups have large class.

`IsMetacyclicPGroup (P)`

P is a p-group, either pc- or matrix or permutation group; if P is metacyclic, then return `true`, else `false`.

`InvariantsMetacyclicPGroup (P)`

P is a metacyclic p-group, either pc- or matrix or permutation group; return tuple of invariants which uniquely identify metacyclic p-group P. This tuple which contains at least four terms, $\langle r, s, t, n \rangle$ has the following meaning: P has order p^{n+s}; its derived quotient is $C_{p^r} \times C_{p^s}$; its derived group is cyclic of order p^{n-r}; it has exponent p^{n+s-t}.

If $p = 2$, then additional invariants are needed to distinguish among the groups. We record the abelian invariants of the centre of P. If $s = 1$ and the centre of P has order 2, then the 2-group is maximal class and we record whether it is dihedral, quaternion or semidihedral. If $s > 1$ then the group has two cyclic central normal subgroups of order 2^{s-1} whose central quotients are both semidihedral, or dihedral and quaternion. The invariant tuple has length at most 6.

`StandardMetacyclicPGroup (P)`

P is a metacyclic p-group, either pc- or matrix or permutation group; return metacyclic p-group having a canonical pc-presentation which is isomorphic to P. If two metacyclic p-groups have the same canonical presentation, then they are isomorphic.

`NumberOfMetacyclicPGroups (p, n)`

Return number of metacyclic groups of order p^n.

`HasAllPQuotientsMetacyclic (G)`

`HasAllPQuotientsMetacyclic (G, p)`

Return `true` if for all primes p all p-quotients of the finitely-presented group G are metacyclic; otherwise return `false` and a description of the set of primes for which G has non-metacyclic p-quotient.

If a prime p is supplied as a second argument, then the function returns `true` if all p-quotients of G are metacyclic; otherwise it returns `false`.

Example H66E6

```
> X := MetacyclicPGroups (3, 6);
> #X;
11
> X[4];
GrpPC of order 729 = 3^6
PC-Relations:
    $.1^3 = $.3,
    $.2^3 = $.4,
    $.3^3 = $.6,
```

```
    $.4^3 = $.5,
    $.5^3 = $.6,
    $.2^$.1 = $.2 * $.6^2
> H := SmallGroup (729, 59);
> IsMetacyclicPGroup (H);
true
> I := InvariantsMetacyclicPGroup(H);
> I;
<2, 2, 2, 4, [], , >
> S := StandardMetacyclicPGroup (H);
GrpPC : S of order 729 = 3^6
PC-Relations:
    S.1^3 = S.3,
    S.2^3 = S.4,
    S.3^3 = S.6,
    S.4^3 = S.5,
    S.5^3 = S.6,
    S.2^S.1 = S.2 * S.6^2
> /* find this group in list */
> [IsIdenticalPresentation (S, X[i]): i in [1..#X]];
[ false, false, false, true, false, false, false, false, false, false, false ]
> /* so this group is #4 in list */
> NumberOfMetacyclicPGroups (19, 7);
14
> Q := FreeGroup (4);
> G := quo < Q |  Q.2^2, Q.4^3, Q.2 * Q.3 * Q.2 * Q.3^-1, Q.1^9>;
> /* are all p-quotients of G metacyclic? */
> HasAllPQuotientsMetacyclic (G);
false [ 3 ]
> /* the 3-quotient is not metacyclic */
```

66.5 Database of Perfect Groups

MAGMA includes a database of finite perfect groups. This database includes all perfect groups up to order 50000, and many classes of perfect groups up to order one million. Each group is defined by means of a finite presentation. Further information is also provided which allows the construction of permutation representations.

66.5.1 Specifying an Entry of the Database

There are three ways to key a particular entry of the database. Firstly, a single integer i simply denotes the i-th entry of the database. There is no particular ordering in the correspondence.

Secondly, the database stores information to quickly locate perfect groups of a particular order; thus the pair o, i represents the i-th entry of order o.

The third method corresponds to the notation used in Chapter 5.3 of [HP89]. In this book, the expression $Q\#p$ denotes the class of groups that are isomorphic to perfect extensions of p-groups by Q, where p is a prime and Q is a fixed finite perfect group in which the largest normal p-subgroup is assumed to be trivial. Within a class $Q\#p$, an isomorphism type of groups is denoted by an ordered pair of integers $\langle r, n\rangle$, where $r \geq 0$ and $n \geq 0$.

To specify a particular group Q without extension, a (somewhat descriptive) string is given. The set of possible values can be accessed using the function `TopQuotients`. Among these strings, full or partial covering groups of G are named G`C`n where n is the index of G in G`C`n. Also, there are five classes of 3-extensions of groups which are also defined in the database. The names of these base groups are `A5#2<`r, n`>` where (r, n) are $(4, 2), (5, 5), (5, 6), (5, 7)$ or $(6, 7)$. Furthermore, there are some extensions of direct products: these have names of the form G`x`H. The remainder are names of simple groups. The convention with these names is that if they are elements of a family of simple groups with two parameters then the name will be *fam*`(`p_1`, `p_2`)`, while one parameter families will just be the concatenation of the family name and the parameter.

To illustrate the naming conventions, here are some examples: `A5`, `A5C2`, `A5#2<`$5, 5$`>`, `L(2, 59)C2`, `A5xL(2, 11)`. Notice that there is never a space before the `C` denoting a covering group or on either side of the `x` denoting direct product. However, there is always a space after a comma.

To specify a particular group $Q\#p\langle r, n\rangle$ the four values Q, p, r and n should be given. However, it should be noted that in three cases (`A5#2< 5, 1>`, `L(3, 2)#2<3, 1>`, `L(3, 2)#2<3, 2>`), there are *two* versions of `Q#p<`r, n`>` stored in the database. Strictly speaking, then, there is a fifth key value v required in this third method. However, it can be specified by an optional parameter `Variant :=` v (if necessary), and can normally be ignored. The variant forms are isomorphic to the original forms, and are included for compatibility with Holt & Plesken's tables.

66.5.2 Creating the Database

`PerfectGroupDatabase()`

This function returns a database object which contains information about the database. It is required as first argument to the other access functions.

66.5.3 Accessing the Database

`Group(D, i)`

`Group(D, o, i)`

`Group(D, Q)`

`Group(D, Q, p, r, n:` *parameters*`)`

`Variant` RngIntElt *Default :* 1

Returns the specified entry from the database D as a finitely presented group. In addition, it returns a sequence of pairs $\langle [i_1, \ldots, i_n], [H_1, \ldots, H_n] \rangle$, each of which affords an isomorphism onto a permutation group of degree $\sum_{j=1}^{n} i_j$. The subgroup H_j has index i_j in the defined group, and the sum of the permutation representations of the group on the cosets of the H_j's is faithful. For the meanings of the arguments, see Subsection 66.5.1 above.

`IdentificationNumber(D, i)`

`IdentificationNumber(D, o, i)`

`IdentificationNumber(D, Q)`

`IdentificationNumber(D, Q, p, r, n:` *parameters*`)`

`Variant` RngIntElt *Default :* 1

Returns a number which can be used to access the specified entry from the database D using method one. (See Subsection 66.5.1 above).

`NumberOfRepresentations(D, i)`

`NumberOfRepresentations(D, o, i)`

`NumberOfRepresentations(D, Q)`

`NumberOfRepresentations(D, Q, p, r, n:` *parameters*`)`

`Variant` RngIntElt *Default :* 1

Returns the number of ways stored in the database for building a permutation group representation of the specified entry. (See Subsection 66.5.1 above).

`PermutationRepresentation(D, i:` *parameters*`)`

`PermutationRepresentation(D, o, i:` *parameters*`)`

`PermutationRepresentation(D, Q:` *parameters*`)`

`PermutationRepresentation(D, Q, p, r, n:` *parameters*`)`

`Variant` RngIntElt *Default :* 1

Returns the isomorphism from the finitely presented group G specified to a permutation group representation H as well as the groups G and H. (See Subsection 66.5.1 above).

`Representation` RngIntElt *Default :* 1

Selects which of the stored methods of constructing the permutation representation should be used.

`PermutationGroup(D, i: parameters)`

`PermutationGroup(D, o, i: parameters)`

`PermutationGroup(D, Q: parameters)`

`PermutationGroup(D, Q, p, r, n: parameters)`

`Variant` RngIntElt *Default :* 1

Returns the specified entry from the database D as a permutation group. (See Subsection 66.5.1 above).

`Representation` RngIntElt *Default :* 1

Selects which of the stored methods of constructing the permutation representation should be used.

66.5.4 Finding Legal Keys

`#D`

`NumberOfGroups(D)`

Returns the number of entries stored in the database. (See Subsection 66.5.1, method 1, above).

`NumberOfGroups(D, o)`

Returns the number of entries stored in the database of order o. (See Subsection 66.5.1, method 2, above).

`TopQuotients(D)`

Returns the set of strings denoting the fixed perfect groups Q. (See Subsection 66.5.1, method 3, above).

`ExtensionPrimes(D, Q)`

Returns the set of primes p for which a non-trivial p-extension of the group denoted by Q lies in the database. (See Subsection 66.5.1, method 3, above).

`ExtensionExponents(D, Q, p)`

Returns the set of exponents r such that a non-trivial extension of the group denoted by Q by p^r lies in the database. (See Subsection 66.5.1, method 3, above).

`ExtensionNumbers(D, Q, p, r)`

Returns the set of numbers n such that there is a group $Q\#p\langle r, n\rangle$ in the database. (See Subsection 66.5.1, method 3, above).

`ExtensionClasses(D, Q)`

Returns the set of triples $\langle p, r, n\rangle$ such that there is a group $Q\#p\langle r, n\rangle$ in the database. (See Subsection 66.5.1, method 3, above).

Example H66E7

We hunt through the various levels of key-finding functions available to find an extension of L(3,4) in the database.

```
> DB := PerfectGroupDatabase();
> "L(3, 4)" in TopQuotients(DB);
true
> ExtensionPrimes(DB, "L(3, 4)");
{ 2 }
> ExtensionExponents(DB, "L(3, 4)", 2);
{ 1, 2, 3, 4 }
> ExtensionNumbers(DB, "L(3, 4)", 2, 2);
{ 1, 2, 3 }
```

The database contains extensions of L(3,4) by groups of order 2^1, 2^2, 2^3 and 2^4. We will look at one of the 3 extensions by a group of order 4.

```
> G := Group(DB, "L(3, 4)", 2, 2, 3);
> G;
Finitely presented group G on 3 generators
Relations
  a^2 = Id(G)
  b^4 * e^-2 = Id(G)
  a * b * a * b * a * b * a * b * a * b * a * b * a * b * e
  = Id(G)
  a * b^2 * a * b^2 * a * b^2 * a * b^2 * a * b^2 * e^-1 =
  Id(G)
  a^-1 * b^-1 * a * b * a^-1 * b^-1 * a * b * a^-1 * b^-1 *
  a * b * a^-1 * b^-1 * a * b * a^-1 * b^-1 * a * b * e^-2 =
  Id(G)
  a * b * a * b * a * b^3 * a * b * a * b * a * b^3 * a * b
  * a * b * a * b^3 * a * b * a * b * a * b^3 * a * b * a *
  b * a * b^3 * e^-2 = Id(G)
  (a * b * a * b * a * b^2 * a * b^-1)^5 = Id(G)
  (a, e^-1) = Id(G)
  (b, e^-1) = Id(G)
> P := PermutationGroup(DB, "L(3, 4)", 2, 2, 3);
> P;
Permutation group P acting on a set of cardinality 224
Order = 80640 = 2^8 * 3^2 * 5 * 7
> ChiefFactors(P);
    G
    |  A(2, 4)               = L(3, 4)
    *
```

```
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
> #Radical(P);
4
> IsCyclic(Radical(P));
true
> IsCentral(P, Radical(P));
true
```

66.6 Database of Almost-Simple Groups

MAGMA includes a database containing information about almost simple groups G, where $S \leq G \leq \mathrm{Aut}(S)$ and S is a simple group of small order. The G that are included in the database are those associated with S such that $|S|$ is less than 16000000, as well as M_{24}, HS, J_3, McL, $Sz(32)$ and $L_6(2)$.

The information stored here is primarily for use in computing maximal subgroups and automorphism groups. The database was originally conceived by Derek Holt, with a major extension by Volker Gebhardt and sporadic additions by Bill Unger. The implementation is by Bruce Cox.

It is possible to request the i-th entry of the database. Alternatively, and more usefully, one can supply three integers: the order $o1$ of S, the order $o2$ of G and the sum k of the orders of the class representatives of G. The last of these can be expensive to compute; however, knowledge of the classes is helpful to benefit from the information stored in the entry. Of course, if the entry is beyond the range of the database, then it is a wasted computation—the intrinsics `ExistsGroupData` and `NumberOfGroups` are provided to determine from the orders whether this is the case.

66.6.1 The Record Fields

The result returned by the `GroupData` function is a record with a number of fields containing information about the almost simple group and its socle. This information includes information used to compute automorphisms and maximal subgroups of the almost simple group by these MAGMA functions. The following describes these fields. The groups G and S are as above the almost simple group G and its socle, the simple group S. Let A be the full automorphism group of S, and let $F\langle x, y\rangle$ be a free group on two generators, called x and y.

First comes information about S. Each S is two generated, by x and y as above, say.

Field `resname`: A string giving a name to the simple group S. (S is the soluble residual of G).

Field `resorder`: The order of S as an integer.

Field `geninfo`: Information on where to find x and y in S. This is a sequence of two tuples, each with 3 entries. The first gives a generator order, the second the length of its conjugacy class, and the third the probability of picking the right generator given the previous information. The first tuple's order/length information always uniquely defines one conjugacy class of the group S and has probability 1, so x is easy to find.

Field `rels`: A sequence of words in F which, taken together with the generator orders from `geninfo`, form a presentation for S on x and y.

Field `permrep`: A permutation representation of the full automorphism group of S. The first two generators are x and y, followed by outer generators. In what follows the outer generators are called t, u, v.

Field `outimages` A sequence of sequences of words in F. These give the images of x and y under the generators of the outer automorphism group of S.

Field `order`: An integer, the order of G.

Field `inv`: The invariant used to separate non-isomorphic $\langle |S|, |G| \rangle$ possibilities. An integer, it is the sum over the classes of G of the order of the elements in each class.

Field `name`: A name for G as a string.

Field `conjelts`: If G is not normal in the automorphism group, these words are coset representatives of the normaliser of G in A as words in t, u, v.

Field `subgens`: Words in t, u, v that, together with x and y, generate G.

Field `subpres`: A presentation of G/S on `subgens`. Note: If $G = S$ then `subgens` will be the empty sequence, but `subpres` will be the trivial FP-group with one generator.

Field `normgens`: Words in t, u, v that generate the outer automorphism group of G.

Field `normpres`: A presentation of the outer automorphism group of G on `normgens`. Again, if $A = G$ then `normgens` will be the empty sequence, but `normpres` will be the trivial FP-group with one generator.

Field `maxsubints`: A sequence of records describing the intersections of the maximal subgroups of G that do not contain S with S. Each record gives the order of the intersection, its class length in S, generators as words in x and y, and a presentation on these generators.

66.6.2 Creating the Database

`AlmostSimpleGroupDatabase()`

This function returns a database object which contains information about the database.

66.6.3 Accessing the Database

`#D`

Returns the number of entries stored in the database.

`GroupData(D, i)`

`GroupData(D, o1, o2, k)`

Returns the specified entry from the database D as a record. The first form gives the ith entry of the database. The second form gives information on an almost simple group of order $o2$, with socle of order $o1$. The value of k should be as explained in `inv` above.

`ExistsGroupData(D, o1, o2)`

`ExistsGroupData(D, o1, o2, i)`

Returns whether any record exists for a simple group of order $o1$ and a supergroup G of order $o2$ lying within its automorphism group. In the second form an invariant, i as described above, is also supplied, and the result is true if there is a record in the database with the given orders and invariant, and false otherwise. When the result is true, the corresponding record is also returned.

`NumberOfGroups(D, o1, o2)`

Returns the number of records in the database D corresponding to a simple group of order $o1$ and a supergroup G of order $o2$ lying within its automorphism group. The second return value gives the index of the first such record, if there is one. (This is most useful when the first return value is 1.) If the return values are d and f, with $d > 0$, then the corresponding database entries are numbered $f, f+1, \ldots, f+d-1$.

`IdentifyAlmostSimpleGroup(G)`

`IdentifyAlmostSimpleGroup(G)`

Use the information in the database to construct a monomorphism f from the almost simple group G into A, the permutation representation of the full automorphism group of its socle stored in the database. This function will also cope with groups isomorphic to the alternating and symmetric groups of degree up to 50, which are not actually in the database. Note that the conjugacy class of the image of f in A determines G up to isomorphism. The algorithms used to deal with the alternating and symmetric groups not in the database are by Derek Holt, starting from the paper of Bratus & Pak [BP00].

Example H66E8

We query the database on about an almost simple group of order 720.

```
> G := PermutationGroup<10 |
>  [ 7, 9, 5, 2, 1, 8, 10, 4, 6, 3 ],
>  [ 6, 3, 10, 7, 2, 4, 1, 8, 9, 5 ]>;
> #G;
720
> CompositionFactors(G);
    G
    |  Cyclic(2)
    *
    |  Alternating(6)
    1
> #Radical(G);
1
```

There are 3 such groups, so we really do need k to tell them apart.

```
> S := SolubleResidual(G);
> k := &+[c[1]: c in Classes(G)];
> D := AlmostSimpleGroupDatabase();
> R := GroupData(D, #S, #G, k);
> R`name;
M_10
```

The group is identified. Let's see some of the other information in R.

```
> P := R`permrep;
> P;
Permutation group P acting on a set of cardinality 10
  (1, 6)(2, 9)(3, 10)(4, 8)
  (1, 7, 9, 4)(2, 8, 5, 10)
  (3, 5, 9, 6, 7, 4, 8, 10)
  (3, 7)(4, 6)(5, 10)
> #P;
1440
> R`subgens;
[ t * u ]
> SS := sub<P|P.1, P.2>;
> #SS;
360
> GG := sub<P|SS, P.3*P.4>;
> #GG;
720
>R`normgens;
[ t ]
```

The full automorphism group of S has order 1440. The group P is a representation of this automorphism group with first two generators generating a faithful image of S. The image of S,

together with the product of P.3 and P.4, generate a faithful image of G. The outer automorphism group of G is generated by P.3 modulo GG. We can also get a constructive identification as follows.

```
> f, A := IdentifyAlmostSimpleGroup(G);
> f;
Homomorphism of GrpPerm: $, Degree 10, Order 2^4 * 3^2 * 5
into GrpPerm: A, Degree 10 induced by
  (1, 10)(3, 8)(5, 9)(6, 7) |--> (1, 6)(2, 9)(3, 10)(4, 8)
  (1, 7, 2, 8)(3, 10, 4, 5) |--> (1, 7, 9, 4)(2, 8, 5, 10)
  (1, 10, 4, 8)(2, 6, 9, 5) |--> (3, 10, 7, 6)(4, 8, 5, 9)
> A;
Permutation group A acting on a set of cardinality 10
  (1, 6)(2, 9)(3, 10)(4, 8)
  (1, 7, 9, 4)(2, 8, 5, 10)
  (3, 5, 9, 6, 7, 4, 8, 10)
  (3, 7)(4, 6)(5, 10)
```

66.7 Database of Transitive Groups

MAGMA has a database containing all transitive permutation groups having degree up to 32, and one containing all primitive permutation groups with degree less than 4096.

The transitive groups up to degree 15 were determined by Greg Butler and John McKay, the groups having degree in the range 16 to 30 were determined by Alexander Hulpke [Hul05]. John Cannon and Derek Holt [CH08] have determined the transitive groups of degree 32.

66.7.1 Accessing the Databases

`TransitiveGroupDatabaseLimit()`

The limiting degree of the database of transitive groups.

`NumberOfTransitiveGroups(d)`

Given a degree d in the required range, return the number of transitive groups of degree d.

`TransitiveGroup(d, n)`

Given a degree d in the required range and a positive integer n, return the n-th transitive group of degree d. Also returns a string giving a description of the group.

`TransitiveGroupDescription(d, n)`

A string giving a description of the n-th transitive group of degree d.

`TransitiveGroupDescription(G)`

A string giving a description of the transitive group G.

`TransitiveGroup(d)`

Given a degree d in the required range, return the first transitive group of degree d. Also returns a string giving a description of the group.

`TransitiveGroup(d, f)`

Given a degree d in the required range and a predicate f (as a function or intrinsic), return the first transitive group of degree d which satisfies f. Also returns a string giving a description of the group.

`TransitiveGroup(S, f)`

Given a sequence S of degrees and a predicate f (as a function or intrinsic), return the first transitive group with degree in S which satisfies f. Also returns a string giving a description of the group.

`TransitiveGroups(d:` *parameters*`)`

`Warning` BoolElt *Default :* `true`

Return a sequence of all transitive groups of degree d. Some degrees will produce a very large sequence of groups – in such cases a warning will be printed unless the user specifies `Warning := false`.

`TransitiveGroups(S:` *parameters*`)`

`Warning` BoolElt *Default :* `true`

Given a sequence S of degrees, return a sequence of all transitive groups with degree in S. The resulting sequence may be very long – in such cases a warning will be printed unless the user specifies `Warning := false`.

`TransitiveGroups(d, f)`

Given an integer d and a predicate (function or intrinsic) f, return a sequence containing all transitive groups G of degree d satisfying `f(G) eq true`.

`TransitiveGroups(S, f)`

Given a sequence S of degrees and a predicate (function or intrinsic) f, return a sequence containing all transitive groups G with degree in S satisfying `f(G) eq true`.

Example H66E9

We apply some of these functions to the degree 8 case.

```
> NumberOfTransitiveGroups(8);
50
> TransitiveGroup(8, 3);
Permutation group acting on a set of cardinality 8
    (1, 2)(3, 4)(5, 6)(7, 8)
    (1, 4)(2, 3)(5, 8)(6, 7)
    (1, 8)(2, 7)(3, 6)(4, 5)
E(8) = 2[x]2[x]2
> S := TransitiveGroups(8, IsPrimitive);
> #S;
7
> S;
[
    Permutation group acting on a set of cardinality 8
        (1, 8)(2, 3)(4, 5)(6, 7)
        (1, 3)(2, 8)(4, 6)(5, 7)
        (1, 5)(2, 6)(3, 7)(4, 8)
        (1, 2, 6, 3, 4, 5, 7),
    Permutation group acting on a set of cardinality 8
        (1, 8)(2, 3)(4, 5)(6, 7)
        (1, 3)(2, 8)(4, 6)(5, 7)
        (1, 5)(2, 6)(3, 7)(4, 8)
        (1, 2, 6, 3, 4, 5, 7)
        (1, 2, 3)(4, 6, 5),
    Permutation group acting on a set of cardinality 8
        (1, 2, 3, 4, 5, 6, 8)
        (1, 2, 4)(3, 6, 5)
        (1, 6)(2, 3)(4, 5)(7, 8),
    Permutation group acting on a set of cardinality 8
        (1, 2, 3, 4, 5, 6, 8)
        (1, 3, 2, 6, 4, 5)
        (1, 6)(2, 3)(4, 5)(7, 8),
    Permutation group acting on a set of cardinality 8
        (1, 8)(2, 3)(4, 5)(6, 7)
        (1, 3)(2, 8)(4, 6)(5, 7)
        (1, 5)(2, 6)(3, 7)(4, 8)
        (1, 2, 6, 3, 4, 5, 7)
        (1, 2, 3)(4, 6, 5)
        (1, 2)(5, 6),
    Permutation group acting on a set of cardinality 8
        (1, 2)(3, 4, 5, 6, 7, 8)
        (1, 2, 3),
    Permutation group acting on a set of cardinality 8
        (1, 2, 3, 4, 5, 6, 7, 8)
        (1, 2)
```

```
]
```

66.7.2 Processes

A transitive group process enables iteration over all transitive groups of specified degrees satisfying a given predicate, without having to create and store all such groups together.

The intrinsic function `TransitiveGroupProcess` may be used to create a transitive group process in MAGMA. The standard process functions `IsEmpty`, `Current`, `CurrentLabel` and `Advance` can then be applied to the process.

`TransitiveGroupProcess(d)`

Return a group process which will iterate though all transitive groups of degree d.

`TransitiveGroupProcess(S)`

Return a process which will iterate though all transitive groups with degree in the sequence S.

`TransitiveGroupProcess(d, f)`

Return a process which will iterate though all transitive groups with degree d which satisfy the predicate f.

`TransitiveGroupProcess(S, f)`

Return a process which will iterate though all transitive groups with degree in the sequence S which satisfy the predicate f.

`IsEmpty(p)`

Returns `true` if the process p has passed its last group.

`Current(p)`

Return the current group of the process p, as well as a description of the group.

`CurrentLabel(p)`

Return the label of the current group of the process p. That is, return d and n such that the current group is `TransitiveGroup(d, n)`.

`Advance(~p)`

Move the process p to its next group.

Example H66E10

The use of processes is illustrated by the following code, in which the orders of all transitive groups of degree 5 are listed.

```
> p := TransitiveGroupProcess(5);
> while not IsEmpty(p) do
>     CurrentLabel(p), #Current(p);
>     Advance(~p);
> end while;
5 1 5
5 2 10
5 3 20
5 4 60
5 5 120
```

66.7.3 Transitive Group Identification

Given a transitive group G whose degree is at most 30, it is possible to obtain the number of the group in the transitive groups database which is isomorphic to G.

`TransitiveGroupIdentification(G)`

`Raw` BoolElt *Default :* `true`

The number (and degree) of the group in the transitive groups database which is isomorphic to the transitive group G.

If the optional parameter `Raw` is set to `false`, a third value is returned. In this case, the third value is a permutation conjugating the given group to the copy in the library.

Example H66E11

We get a transitive permutation group from the small groups database and identify it as a transitive group.

```
> G := SmallGroup(336, IsTransitive: Search:="Insoluble");
> G;
Permutation group G acting on a set of cardinality 16
  (1, 14, 6, 2, 12, 8, 13, 7)(3, 15, 10, 5, 16, 9, 4, 11)
  (2, 5, 6)(3, 10, 9)(4, 15, 16)(7, 11, 13)
> TransitiveGroupIdentification(G : Raw := false);
715 16 (1, 16, 3, 4, 2, 11, 5, 6, 9, 8, 13, 10)(7, 12, 15)
> n, d, p := $1;
> G^p eq TransitiveGroup(d, n);
true
```

We found it to be group 715 of degree 16.

66.8 Database of Primitive Groups

MAGMA has a database containing all primitive permutation groups with degree less than 2500.

The list of primitive groups up to degree 50 was prepared by C. C. Sims (see [Sim70] for the early part of the list). The list up to degree 999 was determined by Roney-Dougal and Unger. See [RDU03] for details of the methods used. The list was extended to degree 2499 by Roney-Dougal, as described in [RD05], and was further extended to degree 4095 by Coutts, Quick and Roney-Dougal [CQRD11].

Within the database the groups are stored by degree. Within each degree they are stored by O'Nan-Scott class in the order soluble affine, insoluble affine, diagonal action, product action, almost simple. Within each class groups are ordered by increasing size. (It follows that the alternating and symmetric groups come last at each degree.)

The basic access function takes two parameters, degree and number, and returns the corresponding primitive group. Functions with name prefixed by `NumberOfPrimitive` tell how many groups of each class there are stored. We recommend the use of the `PrimitiveGroupProcess` or `PrimitiveGroups` functions, with appropriate `Filter` value, to access all primitive groups in a specific class.

66.8.1 Accessing the Databases

`PrimitiveGroupDatabaseLimit()`

The limiting degree of the database of primitive groups.

`NumberOfPrimitiveGroups(d)`

`NumberOfPrimitiveSolubleGroups(d)`

`NumberOfPrimitiveAffineGroups(d)`

`NumberOfPrimitiveDiagonalGroups(d)`

`NumberOfPrimitiveProductGroups(d)`

`NumberOfPrimitiveAlmostSimpleGroups(d)`

Given a degree d in the required range, `NumberOfPrimitiveGroups` returns the number of primitive groups of degree d. The other functions return the number of groups of each class at that degree.

`PrimitiveGroup(d, n)`

Given a degree d in the required range and a positive integer n, return the n-th primitive group of degree d. Also returns a string (possibly empty) giving a description of the group and a string giving the group's O'Nan-Scott type.

`PrimitiveGroupDescription(d, n)`

A string giving a description of the n-th primitive group of degree d.

`PrimitiveGroup(d)`

Given a degree d in the required range, return the first primitive group of degree d. Also returns a string giving a description of the group and a string giving the group's O'Nan-Scott type.

`PrimitiveGroup(d, f)`

Given a degree d in the required range and a predicate f (as a function or intrinsic), return the first transitive (primitive) group of degree d which satisfies f.

`PrimitiveGroup(S, f)`

Given a sequence S of degrees and a predicate f (as a function or intrinsic), return the first transitive (primitive) group with degree in S which satisfies f.

`PrimitiveGroups(d:` *parameters*`)`

`Filter` MonStgElt *Default :* "*All*"

Return a sequence of all primitive groups of degree d, modified by the value assigned to `Filter`. The possible values for the parameter are the strings `All`, `Soluble`, `Affine`, `Diagonal`, `Product`, `AlmostSimple`, `Simple` and `SimpleNA`. Generally these values restrict the list to groups in the appropriate O'Nan-Scott type, with the exceptions being `All` giving no restriction, `Simple` restricting to a lsit of all simple groups in the database, and `SimpleNA` being as for `Simple` but omitting all alternating groups in their natural representations.

`PrimitiveGroups(S:` *parameters*`)`

`PrimitiveGroups( :` *parameters*`)`

`Filter` MonStgElt *Default :* "*All*"

Given a sequence S of degrees, return a sequence of all primitive groups with degree in S. The result is modified by `Filter` with values as above. Omitting the sequence of degrees gives the same result as specifying all legal degrees.

`PrimitiveGroups(d, f:` *parameters*`)`

`PrimitiveGroups(S, f)`

`PrimitiveGroups(f)`

`Filter` MonStgElt *Default :* "*All*"

Given an integer d and a predicate (function or intrinsic) f, return a sequence containing all primitive groups G of degree d passing the filter satisfying `f(G) eq true`. Note that the filter will be generally much quicker in rejecting candidates than the predicate will be, and only groups passing the filter have `f(G)` evaluated.

Instead of giving a single degree, a sequence of degrees may be given. Omitting the degree is the same as specifying the sequence of all legal degrees.

Example H66E12

We apply some of these functions to the degree 625 case.

```
> NumberOfPrimitiveGroups(625);
698
> NumberOfPrimitiveAffineGroups(625);
647
> NumberOfPrimitiveSolubleGroups(625);
509
> NumberOfPrimitiveDiagonalGroups(625);
0
> NumberOfPrimitiveProductGroups(625);
49
> NumberOfPrimitiveAlmostSimpleGroups(625);
2
> PrimitiveGroup(625, 511);
Permutation group acting on a set of cardinality 625
Order = 150000 = 2^4 * 3 * 5^5
5^4:SL(2, 5).2 Affine
> PrimitiveGroup(625,690);
Permutation group acting on a set of cardinality 625
Order = 2^14 * 3^5 * 5^4
Alt(5)^4:Q_8:Sym(4) ProductAction
> Q := PrimitiveGroups(625, func<G|#G eq 3*10^4>
>    : Filter := "Affine");
> #Q;
26
```

66.8.2 Processes

A primitive group process enables iteration over all primitive groups of specified degrees satisfying a given predicate, without having to create and store a list of all such groups.

The intrinsic function `PrimitiveGroupProcess` may be used to create a primitive group process. The standard process functions `IsEmpty`, `Current`, `CurrentLabel` and `Advance` can then be applied to the process.

`PrimitiveGroupProcess(d:` *parameters*`)`

`PrimitiveGroupProcess(S:` *parameters*`)`

`PrimitiveGroupProcess(:` *parameters*`)`

`Filter` MonStgElt *Default :* "*All*"

Return a group process which will iterate though all primitive groups of degree d that pass the filter as described above. A sequence of degrees may be given instead of a single degree. In this case the process will iterate though the groups of all the degrees in S. Omitting any degree information is the same as specifying the sequence of all legal degrees.

`PrimitiveGroupProcess(d, f: parameters)`

`PrimitiveGroupProcess(S, f: parameters)`

`PrimitiveGroupProcess(f: parameters)`

`Filter` MonStgElt *Default : "All"*

Return a process which will iterate though all primitive groups with degree d which pass the filter and satisfy the predicate f. A sequence of degrees may be given instead of a single degree. In this case the process will iterate though the groups of all the degrees in S. Omitting any degree information is the same as specifying the sequence of all legal degrees.

`IsEmpty(p)`

Returns `true` if the process p has passed its last group.

`Current(p)`

Return the current group of the process p, as well as a description of the group.

`CurrentLabel(p)`

Return the label of the current group of the process p. That is, return d and n such that the current group is `TransitiveGroup(d, n)` (or `PrimitiveGroup(d, n)`).

`Advance(~p)`

Move the process p to its next group.

Example H66E13

The use of processes is illustrated by the following code, in which the orders of all primitive groups with degree 60 of diagonal type are listed. We also compute the orbit structures of their Sylow 2-subgroups, which demonstrates that they are non-conjugate.

```
> p := PrimitiveGroupProcess(60:Filter:="Diagonal");
> while not IsEmpty(p) do
>     G := Current(p);
>     CurrentLabel(p), #G,
>     [t[1]:t in OrbitRepresentatives(Sylow(G,2))];
>     Advance(~p);
> end while;
60 1 3600 [ 4, 4, 4, 16, 16, 16 ]
60 2 7200 [ 4, 4, 4, 16, 32 ]
60 3 7200 [ 4, 8, 16, 32 ]
60 4 7200 [ 4, 8, 16, 16, 16 ]
60 5 14400 [ 4, 8, 16, 32 ]
```

66.8.3 Primitive Group Identification

Given a primitive group G whose degree is at most 2499, it is possible to obtain the number of the group in the primitive groups database which is permutation isomorphic to G.

```
PrimitiveGroupIdentification(G)
```

The number (and degree) of the group in the primitive groups database which is permutation isomorphic to the primitive group G.

Example H66E14

We construct a permutation group of affine type and identify it as a primitive group.

```
> M := WreathProduct(SL(2,5), Sym(2));
> Q := Getvecs(M);
> G := Semidir(M, Q);
> G;
Permutation group G acting on a set of cardinality 625
> PrimitiveGroupIdentification(G);
595 625
```

We found it to be group 595 of degree 625.

66.9 Database of Rational Maximal Finite Matrix Groups

MAGMA includes a database of rational maximal finite matrix groups and their invariant forms, for small dimensions (up to 31 at V2.8 and above). This section defines the interface to that database. See the articles of Nebe & Plesken [NP95] and Nebe [Neb96].

A particular entry of the database can be specified in one of two ways. Firstly, a number in the range 1 to the size of the database can be given. Alternatively, the desired dimension can be provided, together with a number in the range 1 to the number of entries of that dimension.

Each entry can be accessed either as a matrix group or as a lattice. If accessed as a matrix group, the order and base are set on return. If as a lattice, the automorphism group is set.

```
RationalMatrixGroupDatabase()
```

This function returns a database object which contains information about the database.

```
LargestDimension(D)
```

Returns the largest dimension of any entry stored in the database. It is an error to refer to larger dimensions in the database.

`#D`

`NumberOfGroups(D)`

`NumberOfLattices(D)`

Returns the number of entries stored in the database.

`NumberOfGroups(D, d)`

`NumberOfLattices(D, d)`

Returns the number of entries stored in the database of dimension d.

`Group(D, i)`

Returns the i-th entry from the database D as a matrix group.

`Lattice(D, i)`

Returns the i-th entry from the database D as a lattice.

`Group(D, d, i)`

Returns the i-th entry of dimension d from the database D as a matrix group.

`Lattice(D, d, i)`

Returns the i-th entry of dimension d from the database D as a lattice.

Example H66E15

```
> D := RationalMatrixGroupDatabase();
> #D;
354
> maxdim := LargestDimension(D);
> maxdim;
31
> &+[ NumberOfGroups(D, d) : d in [ 1 .. maxdim ] ];
354
```

These numbers agree (which is nice). The dimension in that range with the most curves is 24.

```
> S := [ NumberOfGroups(D, d) : d in [ 1 .. maxdim ] ];
> Max(S);
65 24
```

The groups have known order, so it is easy to find the group with smallest order and dimension 24.

```
> time orders := [#Group(D, 24, i): i in [1 .. NumberOfGroups(D, 24)]];
Time: 0.480
> Min(orders);
1872 53
```

66.10 Database of Integral Maximal Finite Matrix Groups

MAGMA includes a database of representatives of the $\mathrm{GL}(n, \mathbf{Z})$-conjugacy classes of irreducible maximal finite subgroups of $\mathrm{GL}(n, \mathbf{Z})$ for $n <= 11$ and $n \in \{13, 17, 19, 23\}$. This section defines the interface to that database.

For $n < 10$ the groups have been described in [PP77, PP80]. The groups of dimension 10 can be found in [Sou94]. In the cases $n > 10$ prime, the representatives have been constructed using the descriptions given in [Ple85].

A particular entry of the database can be specified in one of two ways. Firstly, a number in the range 1 to the size of the database can be given. Alternatively, the desired dimension can be provided, together with a number in the range 1 to the number of entries of that dimension.

Each entry can be accessed either as a matrix group or as a lattice. If accessed as a matrix group, the order and base are set on return. If as a lattice, the automorphism group is set.

`IntegralMatrixGroupDatabase()`

This function returns a database object which contains information about the database.

`LargestDimension(D)`

Returns the largest dimension of any entry stored in the database. It is an error to refer to larger dimensions in the database.

`#D`
`NumberOfGroups(D)`
`NumberOfLattices(D)`

Returns the number of entries stored in the database.

`NumberOfGroups(D, d)`
`NumberOfLattices(D, d)`

Returns the number of entries stored in the database of dimension d.

`Group(D, i)`

Returns the i-th entry from the database D as a matrix group.

`Lattice(D, i)`

Returns a lattice L and sequence of additional forms F fixed by the i-th group in the database D.

`Construction(D, i)`

Returns a string S which describes the construction of the i-th group G in the database D.

If the G-invariant lattice is well known, S equals the name of this lattice. If the Degree d of G is a prime, G usually can be chosen to fix the form $a_0 I_d + a_1(z + z^{-1}) + ... + a_k(z^k + z^{-1})$ with $k = (d-1)/2$ and some $a_i \in \mathbf{Z}$ where z denotes the permutation matrix of some d-cycle in $\mathbf{Z}^{d\times d}$ (see [Ple85]). In this case, S equals $[a_0, a_1, a_2, ...]$. In all other cases, S describes the isomorphism type of G.

The second return value gives the numbers of all groups of degree d in the `Rational Matrix Group Database` which contain a $\mathrm{GL}(d, \mathbf{Q})$-conjugate copy of G.

`Group(D, d, i)`

Returns the i-th entry of dimension d in the database D as a matrix group.

`Lattice(D, d, i)`

Returns a lattice L and sequence of additional forms F fixed by the i-th group of dimension d in the database D.

`Construction(D, d, i)`

Returns a string and integer which describe the construction of the i-th entry of dimension d in the database D.

Example H66E16

```
> D:= IntegralMatrixGroupDatabase();
> #D;
222
> G:= Group(D, 8, 7); Construction(D, 8, 7);
A8* [ 3 ]
```

So G is the automorphism group of the dual of the root lattice A_8 and it is conjugate to a subgroup of the third entry of dimension 8 in the `RationalMatrixgroupDatabase`. We find an explicit embedding T of G into that group.

```
> DQ:= RationalMatrixGroupDatabase();
> H:= Group(DQ, 8, 3); L:= Lattice(DQ, 8, 3);
> F:= PositiveDefiniteForm(G);
> for s in Sublattices(G) do
>   B:= BasisMatrix(s);
>   FF:= B * F * Transpose(B);
>   ok, T:= IsIsometric(LatticeWithGram(FF div GCD(Eltseq(FF))), L);
>   if ok then break; end if;
> end for;
> assert ok;
> T:= Matrix(Rationals(), T*B);
> [Matrix(Integers(), T*Matrix(G.i)*T^-1) in H : i in [1..Ngens(G)]];
[ true, true ]
```

66.11 Database of Finite Quaternionic Matrix Groups

MAGMA includes a database of the finite absolutely irreducible subgroups of $\mathrm{GL}_n(\mathcal{D})$ where $\mathcal{D}$ is a definite quaternion algebra whose centre has degree d over $\mathbf{Q}$ and $nd \leq 10$. This collection is due to Gabriele Nebe [Neb98]. This section defines the interface to that database.

A particular entry of the database can be specified in one of two ways. Firstly, a number in the range 1 to the size of the database can be given. Alternatively, the desired dimension can be provided, together with a number in the range 1 to the number of entries of that dimension.

Each entry can be accessed either as a matrix group or as a lattice. If accessed as a matrix group, the order and base are set on return.

`QuaternionicMatrixGroupDatabase()`

This function returns a database object which contains information about the database.

`LargestDimension(D)`

Returns the largest dimension of any entry stored in the database. It is an error to refer to larger dimensions in the database.

`#D`
`NumberOfGroups(D)`
`NumberOfLattices(D)`

Returns the number of entries stored in the database.

`NumberOfGroups(D, d)`
`NumberOfLattices(D, d)`

Returns the number of entries stored in the database of dimension d.

`Group(D, i)`

Returns the i-th entry from the database D as a matrix group.

`Lattice(D, i)`

Returns a lattice L and sequence of forms F corresponding to the i-th entry of the database D.

`Construction(D, i)`

Returns a string and integer which describe the construction of the i-th entry of the database D.

`Group(D, d, i)`

Returns the i-th entry of dimension d in the database D as a matrix group.

`Lattice(D, d, i)`

Returns a lattice L and sequence of forms F corresponding to the i-th entry of dimension d in the database D.

`Construction(D, d, i)`

Returns a string and integer which describe the construction of the i-th entry of dimension d in the database D.

Example H66E17

We illustrate accessing the quaternionic matrix groups database with a group and lattice of dimension 36.

```
> DB := QuaternionicMatrixGroupDatabase();
> LargestDimension(DB);
40
> NumberOfGroups(DB, 36);
10
> G := Group(DB, 36, 8);
> G : Minimal;
MatrixGroup(36, Integer Ring) of order 43545600 = 2^10 * 3^5
* 5^2 * 7
> #pCore(G, 2);
2
> L, forms := Lattice(DB, 36, 8);
> Determinant(L);
3874204890000
> IsSquare($1);
true 1968300
```

66.12 Database of Finite Symplectic Matrix Groups

MAGMA includes a database of the maximal finite irreducible subgroups of $\mathrm{Sp}_{2n}(\mathbf{Q})$ for $1 \leq i \leq 11$ up to conjugacy in $\mathrm{GL}_{2n}(\mathbf{Q})$. This collection is due to Markus Kirschmer [Kir09]. This section defines the interface to that database.

To avoid non-integral entries, the stored matrix groups do not fix the standard skewsymmetric form but some other nondegenerate skewsymmetric form. The example below illustrates how to construct a conjugate matrix group which fixes the standard skewsymmetric form.

A particular entry of the database can be specified in one of two ways. Firstly, a number in the range 1 to the size of the database can be given. Alternatively, the desired dimension can be provided, together with a number in the range 1 to the number of entries of that dimension.

Each entry can be accessed either as a matrix group or as a lattice with a pair of forms. If accessed as a matrix group, the order and base are set on return.

`SymplecticMatrixGroupDatabase()`

This function returns a database object which contains information about the database.

`LargestDimension(D)`

Returns the largest dimension of any entry stored in the database. It is an error to refer to larger dimensions in the database.

`#D`

`NumberOfGroups(D)`

`NumberOfLattices(D)`

Returns the number of entries stored in the database.

`NumberOfGroups(D, d)`

`NumberOfLattices(D, d)`

Returns the number of entries stored in the database of dimension d.

`Group(D, i)`

Returns the i-th entry from the database D as a matrix group.

`Lattice(D, i)`

Returns a lattice L and a sequence S of two integral forms such that the automorphism group of L with respect to S equals `Group(DB, i)`. The first form in S is the gram matrix of L and the second form is skewsymmetric. The sequence S is normalized as described in the appendix of [Kir09] to simplify the recognition of the matrix group.

`Construction(D, i)`

Returns a string which describes the construction of the i-th entry of the database D.

`Group(D, d, i)`

Returns the i-th entry of dimension d in the database D as a matrix group.

`Lattice(D, d, i)`

Returns a lattice L and a sequence S of forms corresponding to the i-th entry of dimension d in the database D.

`Construction(D, d, i)`

Returns a string which describes the construction of the i-th entry of dimension d in the database D.

Example H66E18

We illustrate accessing the symplectic matrix group database with a group of dimension 16.

```
> DB := SymplecticMatrixGroupDatabase();
> NumberOfGroups(DB, 16);
91
> G := Group(DB, 16, 1);
> G : Minimal;
MatrixGroup(16, Integer Ring) of order 2^21 * 3^4 * 5^2
```

The group G does not fix the standard skewsymmetic form. But it can be conjugated to do so.

```
> _, S := Lattice(DB, 16, 1);
> T := TransformForm(Matrix(Rationals(), S[2]), "symplectic");
> H := ChangeRing(G, Rationals())^(GL(16,Rationals()) ! T);
> J := SymplecticForm(16, Rationals());
> forall{h: h in Generators(H) | h * J * Transpose(h) eq J};
true
```

66.13 Database of Irreducible Matrix Groups

MAGMA has a database containing all irreducible subgroups of $\mathrm{GL}_k(p)$, for p prime, $k \geq 1$ and $p^k < 2500$. One representative of each conjugacy class of subgroups is stored.

The data used is the same as that used to store the affine primitive permutation groups. See the Primitive Groups Database section for the provenance of the data.

Within the database the groups are stored according to p^k. First are the soluble groups, followed by the insoluble. Within each subdivision, the groups are stored by increasing order. (It follows that $\mathrm{GL}_k(p)$ is the last in each list.)

The basic access function takes three parameters, k, p and number, and returns the corresponding group. Functions with name prefixed by `NumberOf` tell how many groups of each class there are stored.

66.13.1 Accessing the Database

`NumberOfIrreducibleMatrixGroups(k, p)`

`NumberOfSolubleIrreducibleMatrixGroups(k, p)`

Given k and p, p prime, $k \geq 1$ and $p^k < 2500$, `NumberOfIrreducibleMatrixGroups` returns the number of subgroups of $\mathrm{GL}_k(p)$ stored. The other function returns the number of soluble subgroups stored.

`IrreducibleMatrixGroup(k, p, n)`

Given k and p p prime, $k \geq 1$ and $p^k < 2500$, and a positive integer n, return the n-th subgroup of $\mathrm{GL}_k(p)$ stored.

Example H66E19

We apply some of these functions to the $\mathrm{GL}_4(5)$ case.

```
> NumberOfIrreducibleMatrixGroups(4, 5);
647
> NumberOfSolubleIrreducibleMatrixGroups(4, 5);
509
> G := IrreducibleMatrixGroup(4, 5, 511);
> ChiefFactors(G);
    G
    |  Cyclic(2)
    *
    |  Alternating(5)
    *
    |  Cyclic(2)
    1
> IsIrreducible(G);
true
> IsAbsolutelyIrreducible(G);
false
```

66.14 Database of Quasisimple Matrix Groups

MAGMA has a database containing characteristic 0 representations of some finite quasisimple groups.

`QuasisimpleMatrixGroup(N, d, p : parameters)`

`OverZ`	BOOLELT	*Default :* `true` $\Leftrightarrow p = 0$
`Automorphisms`	BOOLELT	*Default :* `false`
`RepNo`	RNGINTELT	*Default :* 1

Return an absolutely irreducible matrix group in characteristic p, which may be a prime number or 0, derived from the reduction modulo p of of an absolutely irreducible representation in charateristic 0 and dimension d of the quasisimple group G with name N. The generators of G used are its standard generators. For those quasisimple groups in the ATLAS-database (Section 66.16), the same names are used as there. Other quasisimple groups are named according to the same conventions.

If there is more than one representation of G in dimension d in the database, then the first such is used by default, and the others can be accessed by using the `RepNo` option.

If the reduction modulo p of the representation is not irreducible, then a random non-trivial irreducible constituent is used. (This behaviour may change in the future.)

For those representations that are not realisable over $\mathbf{Z}$ in dimension d, a representation in dimension d over a minimal extension of the rationals and also an irreducible representation in a higher dimension over $\mathbf{Z}$ are both stored in the database. The representation used is the one over $\mathbf{Z}$ if the parameter `OverZ` is `true`, and the one over the number field otherwise. Reduction modulo p is generally faster using the integral representation, so that is the default when $p > 0$.

If the parameter `Automorphisms` is set, then extra generators inducing those outer automorphisms of G that stabilise the representation are included in the group returned. This may result in extra scalars being present in the group returned, and when $p = 0$ this scalar subgroup can sometimes be infinite.

`QuasisimpleMatrixGroups()`

Returns a list of tuples specifying the names of the groups in the quasisimple matrix group database, together with the dimension and the number of stored representations of the group in that dimension.

66.15 Database of Soluble Irreducible Groups

This database contains one representative of each conjugacy class of irreducible soluble subgroups of $\mathrm{GL}(n, p)$, p prime, They may be accessed through specifying a group by its label in the database, as described in the section on basic functions, or through searching using predicates, or through a process. The database was constructed by Mark Short [Sho92].

66.15.1 Basic Functions

The basic access functions for the database are described in this section. The label of a group in the database is three integers, d, p, i. The first, $d \geq 2$, is the degree of the matrix group. The second, a prime p, specifies the base field of the group. The third is the number of the group in this degree/field set.

`IsolGroupDatabase()`

Open the database and return a reference to it. This reference may be passed to other functions so that they do fewer file operations.

`IsolGroup(n, p, i)`

`Group(D, n, p, i)`

Given a positive integer $o \leq 1000$ (with $o \neq 512$ or 768) and a positive integer n, return the n-th group of order o.

`IsolNumberOfDegreeField(n, p)`

The number of groups in the database of degree n over $\mathbf{F}_p$.

`IsolInfo(n, p, i)`

This function returns a string which gives some information about a group in the database given its label. In particular, it contains the order and primitivity information about the group.

`IsolOrder(n, p, i)`

This function returns the order of a group given its label.

`IsolMinBlockSize(n, p, i)`

This function returns the minimal block size of a group given its label. If it is primitive, it returns 0.

`IsolIsPrimitive(n, p, i)`

This function returns whether a group is primitive given its label.

`IsolGuardian(n, p, i)`

This function returns the "guardian" of a group given its label, i.e., the maximal subgroup of $\mathrm{GL}(n, p)$ of which the group is a subgroup.

Example H66E20

We find a group of degree 3 and its guardian.

```
> IsolNumberOfDegreeField(3, 5);
22
> G := IsolGroup(3, 5, 10);
> #G;
62
> GG := IsolGuardian(3, 5, 10);
> #GG;
372
> G;
MatrixGroup(3, GF(5)) of order 62 = 2 * 31
Generators:
  [0 0 1]
  [3 0 4]
  [2 3 1]
> GG;
MatrixGroup(3, GF(5)) of order 372 = 2^2 * 3 * 31
Generators:
  [1 0 0]
  [3 2 2]
  [1 4 2]

  [0 1 0]
  [0 0 1]
  [3 0 4]
```

66.15.2 Searching with Predicates

We may search the database for a group satisfying some predicate. A predicate for a group in this database is one of the following:

- A function f (which may either be an intrinsic function or a user defined function) which takes a matrix group and returns a boolean value.
- A tuple of one function $\langle g \rangle$, where g takes a label and returns a boolean value. Again g is either intrinsic or user defined.
- A tuple of two functions $\langle g, f \rangle$ where g, f are as above. In this case, the tested predicate will be g first, then f. This form is introduced to avoid expanding the group from its label until absolutely necessary.

`IsolGroupSatisfying(f)`

Given a predicate f, return a group satisfying it. This function runs through all the stored groups and applies the predicate until it finds a suitable one. If no group is found, an error message is printed.

`IsolGroupOfDegreeSatisfying(d, f)`

As `IsolGroupSatisfying(f)`, except it only runs through the groups of degree d.

`IsolGroupOfDegreeFieldSatisfying(d, p, f)`

As `IsolGroupSatisfying(f)`, except it only runs through the groups of degree d and defined over $\mathbf{F}_p$.

`IsolGroupsSatisfying(f)`

As `IsolGroupSatisfying(f)`, except a sequence of all such groups is returned.

`IsolGroupsOfDegreeSatisfying(d, f)`

As `IsolGroupOfDegreeSatisfying(d, f)`, except a sequence of all such groups is returned.

`IsolGroupsOfDegreeFieldSatisfying(d, p, f)`

As `IsolGroupOfDegreeFieldSatisfying(d, p, f)`, except a sequence of all such groups is returned.

66.15.3 Associated Functions

Associated with this database are two functions useful for constructing semidirect product of a finite vector space and an irreducible matrix group. Thus for constructing soluble affine permutation groups.

`Getvecs(G)`

This function takes a matrix group G over a finite prime field and returns a sequence, Q say, containing all the vectors of the natural module for G. The ordering of Q does *not* depend on G, but only on its natural module.

`Semidir(G, Q)`

Given an irreducible matrix group G of degree n and over a finite prime field of size p and the sequence Q obtained from `Getvecs`, this function returns the permutation group H of degree p^n that is the semidirect product of G with its natural module. H acts on the set $\{1 \dots p^n\}$ and G is isomorphic to each of the point stabilizers. It is well known that H is primitive, and that every primitive permutation group with soluble socle arises in this way. Note that if `Semidir` is to be called more than once for subgroups of the same general linear group, then `Getvecs` need only be called on the first occasion, since the ordering of Q depends only on n and p. This is why the call to `Getvecs` is not made by `Semidir` itself.

66.15.4 Processes

A small group process enables iteration over all groups with specified degrees and fields, without having to create and store all such groups together.

A process is created via the function `IsolProcess` and its variants. The standard process functions `IsEmpty`, `Current`, `CurrentLabel` and `Advance` can then be applied to the process.

A specifier for degree or field is one of a valid degree (field size), or a tuple $\langle l, h \rangle$, of valid degrees (field sizes) which is interpreted to mean all degrees (prime field sizes) in $[l, h]$.

`IsolProcess()`

Return a process which will iterate though all groups in the database.

`IsolProcessOfDegree(d)`

Return a process which will iterate though all groups in the database of degree d.

`IsolProcessOfField(p)`

Return a process which will iterate though all groups in the database over the specified field.

`IsolProcessOfDegreeField(d, p)`

Return a process for iterating over all the stored groups with degree specifier d and field specifier p. Initially it points to the first such group (the principal key is the degree).

`IsEmpty(p)`

Returns `true` if the process p has passed its last group.

`Current(p)`

Return the current group of the process p.

`CurrentLabel(p)`

Return the label of the current group of the process p. That is, return d, n and i such that the current group is `IsolGroup(d, n, i)`.

`Advance(~p)`

Move the process p to its next group.

Example H66E21

We use a small group process to look at all the groups of degree 3.

```
> P := IsolProcessOfDegree(3);
> ords := {* *};
> repeat
>   Include(~ords, #Current(P));
>   Advance(~P);
> until IsEmpty(P);
> ords;
{* 31, 62, 93, 7, 124, 96^^4, 39, 12^^2, 186, 13, 192^^2, 48^^4,
21, 24^^6, 26 *}
```

66.16 Database of ATLAS Groups

MAGMA includes representations of nearly simple groups from the ATLAS of Finite Group Representations http://web.mat.bham.ac.uk/atlas/v2.0. The data was supplied by Robert Wilson.

Groups in the database are accessed by name. The intrinsic ATLASGroupNames gives a list of the names that may currently be used to access the database. The names are based on ATLAS names for simple groups, with some exceptions (usually caused by an aversion to subscripting automorphisms). Classical group names take precedence over their Lie-type names. Within a name, the letter "T" denotes a twisted group of Lie type. (The two sorts of twisting of D_4 are distinguished by one being "O8m" and the other "TD4".) An initial number on the name denotes a central element, a "d" is used to separate the simple group name from an automorphism (when there is no other letter there), and an "i" denotes an isoclinic variant.

Example H66E22

The list of names in V2.11 is printed as follows.

```
> ATLASGroupNames();
{@ A5, 2A5, 2S5, 2S5i, S5, A6, 2A6, 2S6, 3A6, 3S6, 6A6,
6S6, A6V4, M10, PGL29, S6, A7, A8, 2A8, S8, A9, 2A9,
S9, A10, 2A10, S10, A11, 2A11, 2S11, S11, A12, 2A12,
S12, A13, 2A13, S13, A14, 2A14, 2S14, 2S14i, S14, O93,
2O93, 2O93d2, O93d2, O10m2, O10m2d2, O73, 2O73, 2O73d2,
3O73, 3O73d2, O73d2, O8m2, O8m2d2, O8m3, 2O8m3,
2O8m3d2a, O8m3D8, O8m3V4, O8m3d2a, O8p2, S102, S44,
S44d2, S44d4, S45, 2S45, S45d2, S47, 2S47, 2S47d2,
S47d2, S62, 2S62, S63, 2S63, 2S63d2, S63d2, S82, U311,
3U311, 3U311d2, U311d2, U33, U33d2, U42, 2U42, 2U42d2,
U42d2, U43, U52, U52d2, U53, U62, 12U62, 2U62, 3U62,
4U62, 6U62, U62S3, U62d2, U72, E74, E85, E82, E72, E62,
TF42, TF42d2, G25, TE62, 2TE62, 2TE62d2, 3TE62,
3TE62S3, 3TE62d2, 3TE62d3, 4TE62, TE62S3, TE62d2,
TE62d3, E64, 3E64, 3E64d2, TD42, TD42d3, G23, 3G23,
3G23d2, G23d2, G24, 2G24, 2G24d2, 2G24d2i, G24d2, F42,
2F42, 2F42d2, 2F42d4i, F42d2, R27, R27d3, Sz8, 2Sz8,
4Sz8d3, Sz8d3, Sz32, Sz32d5, TD43, L27, L28, L28d3,
L211, 2L211, L211d2, L213, 2L213, 2L213d2, L213d2,
L216, L216d2, L216d4, L217, 2L217, 2L217d2, L217d2,
L219, 2L219, 2L219d2i, L219d2, L223, 2L223, 2L223d2i,
L223d2, L227, L229, 2L229, L231, 2L231, L231d2, L232,
L232d5, L249, 2L249, L33, L33d2, L34, 12aL34, 12bL34,
2L34, 3L34, 4aL34, 4bL34, 6L34, L35, L35d2, L37, 3L37,
3L37d2, L37d2, L311, L52, L52d2, L62, L62d2, L72,
L72d2, B, Co1, 2Co1, Co2, Co3, F22, 2F22, 2F22d2, 3F22,
3F22d2, F22d2, F23, F24, 3F24, 3F24d2, F24d2, HN, HNd2,
HS, 2HS, 2HSd2, HSd2, He, Hed2, J1, J2, 2J2, 2J2d2,
```

```
J2d2, J3, 3J3, 3J3d2, J3d2, J4, Ly, ON, 3ON, 3ONd2,
ONd2, ONd4, Ru, 2Ru, Suz, 2Suz, 2Suzd2, 3Suz, 3Suzd2,
6Suz, 6Suzd2, Suzd2, Th, M, M11, M12, 2M12, 2M12d2,
M12d2, M22, 12M22, 2M22, 2M22d2, 3M22, 3M22d2, 4M22,
4M22d2, 6M22, 6M22d2, M22d2, M23, M24, McL, 3McL,
3McLd2, McLd2, S7 @}
```

The basic access function takes a name and returns a special type of group, an ATLAS group, with MAGMA type `GrpAtlas`. Access to the information stored about the named group are then done through this ATLAS group.

66.16.1 Accessing the Database

`ATLASGroupNames()`

The names of the groups that have representations stored in the database.

`ATLASGroup(N)`

The ATLAS group stored in the database that has name N.

66.16.2 Accessing the ATLAS Groups

Once an ATLAS group has been extracted from the database, the following intrinsics give access to the information stored with it.

`Order(A)`
`#G`

The order of A.

`Multiplier(A)`

The order of the multiplier of A, when A is simple.

`MatRepKeys(A)`

The sequence of keys to the matrix representations of A stored in the database. This will be the empty sequence if no matrix representations are stored.

`MatRepDegrees(A)`

The set of degrees of the matrix representations stored for A.

`MatRepFieldSizes(A)`

The set of sizes of the fields for which a matrix representation of A is available.

`MatRepCharacteristics(A)`

The set of characteristics of the fields for which a matrix representation of A is available.

`PermRepKeys(A)`

The sequence of keys to the permutation representations of A stored in the database. This will be the empty sequence if no permutation representations are stored.

`PermRepDegrees(A)`

The set of degrees of the permutation representations stored for A.

66.16.3 Representations of the ATLAS Groups

The intrinsics described below construct concrete representations of the ATLAS groups from the data in the database. Each representation is accessed by its key, sequences of which are produced by the intrinsics `MatRepKeys` and `PermRepKeys` described above. The intrinsics described in this section take a key and produce a concrete representation.

`MatrixGroup(K)`

Given a key to a matrix representation of an ATLAS group, construct and return the corresponding matrix group.

`MatRep(K)`

The generators of the matrix group designated by database key K.

`PermutationGroup(K)`

Given a key to a permutation representation of an ATLAS group, construct and return the corresponding permutation group.

`PermRep(K)`

The generators of the permutation group designated by database key K.

Example H66E23

We get a representation of $2.J_2.2$ from the database.

```
> A := ATLASGroup("2J2d2");
> PermRepKeys(A);
[]
> mrk := MatRepKeys(A);
> mrk;
[
  Matrix rep of degree 12 over GF(3),
  Matrix rep of degree 6 over GF(25) named a,
  Matrix rep of degree 12 over GF(7)
]
```

The database has no permutation representations and three matrix representations. We construct the first of the matrix groups. It is small enough to check its composition factors.

```
> K := mrk[1];
> M := MatrixGroup(K);
```

```
> M`Order := #A;
> RandomSchreier(M);
> CompositionFactors(M);
    G
    |  Cyclic(2)
    *
    |  J2
    *
    |  Cyclic(2)
    1
```

For efficiency, we asserted the order of the matrix group to be the order of the ATLAS group and constructed a BSGS by the random schreier.

66.17 Fundamental Groups of 3-Manifolds

The database consists of the fundamental groups of the 10,986 small-volume closed hyperbolic manifolds in the Hodgson-Weeks census. The presentations included were generated by Jeffrey Weeks' program *SnapPea* `http://www.geometrygames.org/SnapPea/`. Information about finite-index subgroups with homology was generated by Dunfield and Thurston in [DT03].

66.17.1 Basic Functions

The basic access functions for the database are described in this section.

The result returned by the `Manifold` function is a record with a number of fields containing information about the manifold and its fundamental group. The fields of the records are as follows:

Field `Name`: A string giving a name to the manifold M.

Field `Volume`: The volume of M as a floating point number.

Field `Homology`: A sequence of integers describing the first homology group of M.

Field `Group`; The fundamental group of M as a finitely presented group.

Field `GoodCoverImage`: A possibly empty sequence of permutations or integers 1 representing the identity permutation. These permutations define a homomorphism from the fundamental group to S_n, such that the kernel of the homomorphism has infinite abelianization.

Field `GoodCover`: A list describing the construction of the good cover.

Field `Degree`: A positive integer, the degree of the `GoodCoverImage` permutation representation.

Field `KnownPosBettiCover`: A boolean value, always true in the current database.

Field `KnownWeakPosBettiCover`: A boolean value, always true in the current database.

Field `Reason`: A string, one of `"AbelianInvariants"`, `"RationalReconstruction"` or `"MAGMA"`.

Field `Rank`: A positive integer.

Field `GoodCoverImageU`: A possibly empty sequence of permutations or integers 1 representing the identity permutation.

`ManifoldDatabase()`

Open the database and return a reference to it.

`Manifold(D, i)`

Extract the ith record from the database of fundamental groups of 3-dimensional manifolds. The current limits on i are $1 \leq i \leq 11126$.

66.17.2 Accessing the Data

The intrinsic `Manifold` is one way to access the data in the database. It may be more convenient to iterate over the database object returned by `ManifoldDatabase`. The following examples show how this may be done.

Example H66E24

We extract a record from the database.

```
> D := ManifoldDatabase();
> r := Manifold(D, 100);
> r`Name;
m019(1,4)
> r`Homology;
[ 2, 31 ]
> r`Group;
Finitely presented group on 2 generators
Relations
  $.1 * $.2^3 * $.1 * $.2 * $.1^4 * $.2 * $.1 * $.2 * $.1^4
    * $.2 = Id($)
  $.1 * $.2 * $.1 * $.2^2 * $.1^-3 * $.2^2 = Id($)
> r`GoodCoverImage;
[
  (1, 2, 4, 6, 5, 8, 7, 9, 3),
  (1, 3, 5, 4, 7, 6, 9, 8, 2)
]
```

In [DT03], Dunfield and Thurston note that they found 132 manifolds with positive Betti number. We find them in the database as those records where the `Degree` is 1. We then search the database for one of these, but by name. Both searches use the facility to iterate over the database that was mentioned above.

```
> D := ManifoldDatabase();
> pos_betti := {r`Name:r in D|r`Degree eq 1};
```

```
> #pos_betti;
132
> Random(pos_betti);
s527(-5,1)
> exists(r){r:r in D|r`Name eq "s527(-5,1)"};
true
> F := r`Group; F;
Finitely presented group F on 2 generators
Relations
  F.1^2 * F.2^2 * F.1^2 * F.2^-1 * F.1^2 * F.2^2 * F.1^2 *
  F.2^2 * F.1^-1 * F.2^2 = Id(F)
  F.1^2 * F.2^2 * F.1^2 * F.2 * F.1^2 * F.2^2 * F.1^2 * F.2
  * F.1^2 * F.2^2 * F.1^2 * F.2 * F.1^2 * F.2^2 * F.1^2 *
  F.2 * F.1^2 * F.2^2 * F.1^2 * F.2 * F.1^2 * F.2^2 * F.1^2
  * F.2^2 * F.1^-3 * F.2^2 = Id(F)
> AbelianQuotientInvariants(F);
[ 7, 0 ]
> r`Homology;
[ 0, 7 ]
```

As expected, we see that the fundamental group has infinite abelianization.

66.18 Bibliography

[**BE99a**] Hans Ulrich Besche and Bettina Eick. Construction of finite groups. *J. Symbolic Comput.*, 27(4):387–404, 1999.

[**BE99b**] Hans Ulrich Besche and Bettina Eick. The groups of order at most 1000 except 512 and 768. *J. Symbolic Comput.*, 27(4):405–413, 1999.

[**BE01**] Hans Ulrich Besche and Bettina Eick. The groups of order $q^n \cdot p$. *Comm. Algebra*, 29(4):1759–1772, 2001.

[**BEO01**] Hans Ulrich Besche, Bettina Eick, and E. A. O'Brien. The groups of order at most 2000. *Electron. Res. Announc. Amer. Math. Soc.*, 7:1–4 (electronic), 2001.

[**BP00**] Sergey Bratus and Igor Pak. Fast constructive recognition of a black box group isomorphic to S_n or A_n using Goldbach's conjecture. *J. Symbolic Comp.*, 29:33–57, 2000.

[**CH08**] J.J. Cannon and D.F. Holt. The transitive permutation groups of degree 32. *Experiment. Math.*, 17:307–314, 2008.

[**CQRD11**] Hannah J. Coutts, Martyn Quick, and Colva M. Roney-Dougal. The primitive permutation groups of degree less than 4096. *Communications in Algebra*, 39:10: 3526–3546, 2011.

[**DE05**] Heiko Dietrich and Bettina Eick. On the groups of cubefree order. *J. Algebra*, 292:122–137, 2005.

[**DT03**] Nathan M. Dunfield and William P. Thurston. The virtual Haken conjecture; experiments and examples. *Geometry & Topology*, 7:399–441, 2003.

[**HP89**] D.F. Holt and W. Plesken. *Perfect Groups*. Oxford University Press, 1989.

[**Hul05**] Alexander Hulpke. Constructing transitive permutation groups. *J. Symbolic Comput.*, 39(1):1–30, 2005.

[**Kir09**] M. Kirschmer. *Finite symplectic matrix groups*. Dissertation, RWTH Aachen, 2009. available at URL:http://www.math.rwth-aachen.de/ Markus.Kirschmer/symplectic/thesis.pdf.

[**MNVL04**] E.A. O'Brien M.F. Newman and M.R. Vaughan-Lee. Groups and nilpotent Lie rings whose order is the sixth power of a prime. *J. Algebra*, 278:383–401, 2004.

[**Neb96**] G. Nebe. Finite subgroups of $\mathrm{GL}_n(\mathbf{Q})$ for $25 \leq n \leq 31$. *Comm. Algebra*, 24(7):2341–2397, 1996.

[**Neb98**] G. Nebe. Finite quaternionic matrix groups. *Represent. Theory*, 2:106–223, 1998.

[**NP95**] G. Nebe and W. Plesken. Finite rational matrix groups. *Mem. Amer. Math. Soc.*, 116(556), 1995.

[**O'B90**] E.A. O'Brien. The p-group generation algorithm. *J. Symbolic Comput.*, 9:677–698, 1990.

[**O'B91**] E.A. O'Brien. The Groups of Order 256. *J. Algebra*, 143:219–235, 1991.

[**OVL05**] E.A. O'Brien and M.R. Vaughan-Lee. The groups with order p^7 for odd prime p. *J. Algebra*, 2005.

[**Ple85**] Wilhelm Plesken. Finite unimodular groups of prime degree and circulants. *J. Algebra*, 97:286–312, 1985.

[**PP77**] Wilhelm Plesken and Michael Pohst. On maximal finite irreducible subgroups of GL(n,Z). Parts I and II. *Math. Comp.*, 31:536–576, 1977.

[**PP80**] Wilhelm Plesken and Michael Pohst. On maximal finite irreducible subgroups of GL(n,Z). Parts III-V. *Math. Comp.*, 34(149):245–301, 1980.

[**RD05**] Colva M. Roney-Dougal. The primitive permutation groups of degree less than 2500. *J. Algebra*, 292(1):154–183, 2005.

[**RDU03**] Colva M. Roney-Dougal and William R. Unger. The affine primitive permutation groups of degree less than 1000. *J. Symbolic Comp.*, 35:421–439, 2003.

[**Sho92**] Mark W. Short. *The Primitive Soluble Permutation Groups of Degree less than 256*, volume 1519 of *Lecture Notes in Math.* Springer, Berlin and Heidelberg, 1992.

[**Sim70**] C.C. Sims. Computational methods in the study of permutation groups. In J. Leech, editor, *Computational problems in abstract algebra*, pages 169–183. Oxford - Pergamon, 1970.

[**Sou94**] Bernd Souvignier. Irreducible finite integral matrix groups of degree 8 and 10. *Math. Comp.*, 63:335–350, 1994.

67 AUTOMORPHISM GROUPS

Chapter 67
AUTOMORPHISM GROUPS

67.1 Introduction

MAGMA provides facilities for constructing and working with automorphism groups of various objects. In this chapter we describe the machinery provided in MAGMA for groups of automorphisms in the case of groups.

An *automorphism* of a group G is a bijective homomorphism from G to itself. The set of all automorphisms of G forms a group U known as the *automorphism group* of G. A subgroup A of U will be referred to as a *group of automorphisms* of G. The group G is called the *base group* of a group of automorphisms A and we say that A acts on G. Each MAGMA automorphism group A stores, as part of its data structure, a generating set for its base group, and each automorphism of A is described by its action on these generators.

The full group of automorphisms may be found using an algorithm that proceeds as follows: A series of characteristic subgroups

$$1 = N_r < N_{r-1} < \ldots < N_1 = L < G$$

is constructed for the given group G, such that each N_i/N_{i+1} is elementary abelian and such that G/L has no non-trivial soluble normal subgroup. The automorphism group is found for each of the associated factor groups of G, starting with the top factor G/L and lifting through each layer N_i/N_{i+1} in turn, until we finally have the automorphism group for G itself. The general algorithm for a non-soluble group is described in Cannon and Holt [CH03]. More specialised versions are described by Eick, Leedham-Green and O'Brien [ELGO02] (p-groups) and Smith [Smi94] (soluble groups). This general class of algorithms will be referred to collectively as *lifting algorithms*.

When G is a non-soluble permutation or matrix group, the algorithm relies on a database of automorphism groups for the non-cyclic simple factors of G, hence the non-abelian composition factors of G must belong to a restricted list. In V2.11 this list includes all simple groups of order at most 1.6×10^7, the alternating groups of degree at most 1000, all groups from several generic families, including $PSL(2, q)$, $PSL(3, q)$, $PSL(4, p)$, $PSL(5, p)$, $PSU(3, p)$ and $PSp(4, p)$ and the sporadic groups M_{11}, M_{12}, M_{22}, M_{23}, M_{24}, J_1, J_2, J_3, HS, McL, $Co3$, He and others. The list is being extended regularly.

An automorphism group A of G is represented as a set of homomorphisms of G into itself. We shall refer to this as the mappings representation of A. The full automorphism group is also returned as a finitely presented group and, in addition, it is also possible to construct a permutation representation of the automorphism.

The family of all groups of automorphisms forms a category. The objects are the automorphism groups and the morphisms are group homomorphisms. The MAGMA designation for this category of automorphism groups is `GrpAuto`.

67.2 Creation of Automorphism Groups

An automorphism group of the finite group G may be created in one of two ways. Firstly, the full automorphism group of G may be constructed by invoking an appropriate lifting algorithm. Secondly, an arbitrary group of automorphisms A of G may be created by giving a set of generators for A defined in terms of their action on a set of generators for G.

`AutomorphismGroup(G)`

Given a finite group G, construct the full automorphism group F of G. The group G may be a permutation group, a (finite) matrix group or a finite soluble group given by a pc-presentation. The function returns the full automorphism group of G as a group of mappings (i.e., as a group of type `GrpAuto`). If G is a permutation or matrix group, then the automorphism group F is also computed as a finitely presented group and can be accessed via the function `FPGroup(F)`. A function `PermutationRepresentation` is provided that when applied to F attempts to construct a faithful permutation representation of reasonable degree (see below).

`SmallOuterAutGroup` RngIntElt *Default :* 20000

`SmallOuterAutGroup := t`: Specify the strategy for the backtrack search when testing an automorphism for lifting to the next layer. If the outer automorphism group O at the previous level has order at most t, then the regular representation of O is used, otherwise the program tries to find a smaller degree permutation representation of O.

`Print` RngIntElt *Default :* 0

The level of verbose printing. The possible values are 0, 1, 2 or 3.

`PrintSearchCount` RngIntElt *Default :* 1000

`PrintSearchCount := s`: If `Print := 3`, then a message is printed at each s-th iteration during the backtrack search for lifting automorphisms.

In the case of a non-soluble group, the algorithm described in Cannon and Holt [CH03] is used. If G is a p-group of type `GrpPC` the algorithm described in Eick, Leedham-Green and O'Brien [ELGO02] is used. For more details see Section 63.12.2. If G is of type `GrpPC` but is not a p-group, the algorithm of Smith [Smi94], as extended by Smith and Slattery, is used. For more details see Section 63.12.

When G is a non-soluble permutation or matrix group, the algorithm relies on a database of automorphism groups for the non-cyclic simple factors of G, hence the non-abelian composition factors of G must belong to a restricted list. In V2.11 this list includes all simple groups of order at most 1.6×10^7, the alternating groups of degree at most 1000, all groups from several generic families, including $PSL(2,q)$, $PSL(3,q)$, $PSL(4,p)$, $PSL(5,p)$, $PSU(3,p)$ and $PSp(4,p)$ and the sporadic groups M_{11}, M_{12}, M_{22}, M_{23}, M_{24}, J_1, J_2, J_3, HS, McL, $Co3$, He and others. The list is being extended regularly.

Example H67E1

We create a non-soluble group G of 4×4 matrices defined over the field of 8-th roots of unity and construct its automorphism group.

```
> L<zeta_8> := CyclotomicField(8);
> i := -zeta_8^2;
> t := zeta_8^3;
> G := MatrixGroup< 4, L |
>             [ 1/2, 1/2, 1/2, 1/2,
>               1/2,-1/2, 1/2,-1/2,
>               1/2, 1/2,-1/2,-1/2,
>               1/2,-1/2,-1/2, 1/2 ],
>             DiagonalMatrix( [1,1,1,-1] ),
>             DiagonalMatrix( [1,i,1,i] ),
>             DiagonalMatrix( [t,t,t,t] ) >;
> Order(G);
92160
> CompositionFactors(G);
    G
    |  Cyclic(2)
    *
    |  Alternating(6)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
> A := AutomorphismGroup(G);
> Order(A);
92160
```

`AutomorphismGroup(G, Q, I)`

Let G be a finite group and let Q be a sequence of elements which generate G. Let $\phi_1, \ldots, \phi_r$ be a sequence of automorphisms of G that generate the group of automorphisms A. The group A is specified by a sequence I of length r where the i-th term of I defines ϕ_i in terms of a sequence containing the images of the elements of Q under the action of ϕ_i. The function returns the group of automorphisms A of G.

67.3 Access Functions

The functions described here provide access to basic information stored for an automorphism group A.

`Group(A)`

Given a group of automorphisms A of the group G, return the base group G on which A acts.

`NumberOfGenerators(A)`

`Ngens(A)`

Given a group of automorphisms A of the group G, return the number of defining generators for A.

`NumberOfPCGenerators(A)`

`NPCGenerators(A)`

`NPCgens(A)`

Given a group of automorphisms A of the group G, where a pc-representation has been created for A (and attribute `PCGenerators` is set on A), return the number of pc-generators for A.

`Generators(A)`

Given a group of automorphisms A of the group G, return a set containing the defining generators of A.

`PCGenerators(A)`

Given a group of automorphisms A of the group G, where a pc-representation has been created for A (and attribute `PCGenerators` is set on A), return an indexed set containing the pc-generators of A.

`InnerGenerators(A)`

Given the full group of automorphisms A of the group G, return a sequence of generators for the inner automorphism group of the base group of A (attribute `InnerGenerators`), if this attribute has been set.

`CharacteristicSeries(A)`

Given a group of automorphisms A of the group G, return the value of the characteristic series of G used to compute A, if this attribute has been set.

`IsSoluble(A)`
`IsSolvable(A)`

Given a group of automorphisms A of the group G, return the value of A's attribute `Soluble`, if this attribute has been set.

`IsSolubleAutomorphismGroupPGroup(A)`
`IsSolvableAutomorphismGroupPGroup(A)`

Given a group of automorphisms A of a p-group G constructed using the intrinsic `AutomorphismGroup(G)` or any equivalent alias, determine if A is soluble and return the result. This function also sets the `Soluble` attribute on A.

67.4 Order Functions

Unless the order is already known, each of the functions in this family will create a faithful permutation representation of the group of automorphisms in order to compute the order.

`Order(A)`
`#A`

The order of the group of automorphisms A, returned as an integer. If not already known, this function will create a permutation representation for A.

`FactoredOrder(A)`

The factored order of the group of automorphisms A. If not already known, this function will create a permutation representation for A.

`OuterOrder(A)`

The order of the outer automorphism group associated with the group of automorphisms A.

Example H67E2

We create the non-soluble group $G = PGL(2, 9)$ and examine the properties of its automorphism group.

```
> G := PGL(2, 9);
> A := AutomorphismGroup(G);
> A;
A group of automorphisms of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5
Generators:
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (1, 7, 3, 5, 4, 2, 10, 9)
```

```
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 6, 8)(2, 7, 10)(3, 9, 5)
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (1, 4, 6, 10, 7, 8, 5, 9)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 9, 10)(2, 6, 3)(4, 8, 7)
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (3, 5, 9, 6, 7, 4, 8, 10)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 10, 2)(3, 4, 7)(5, 8, 9)
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (1, 10, 3, 5, 2, 4, 7, 6)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 6, 2)(3, 7, 5)(4, 9, 8)
> #A;
1440
> FactoredOrder(A);
[ <2, 5>, <3, 2>, <5, 1> ]
> OuterOrder(A);
2
> InnerGenerators(A);
[
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (1, 7, 3, 5, 4, 2, 10, 9)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 6, 8)(2, 7, 10)(3, 9, 5),
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (1, 4, 6, 10, 7, 8, 5, 9)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 9, 10)(2, 6, 3)(4, 8, 7),
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (3, 5, 9, 6, 7, 4, 8, 10)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 10, 2)(3, 4, 7)(5, 8, 9)
]
> CharacteristicSeries(A);
[
    Permutation group G acting on a set of cardinality 10
    Order = 720 = 2^4 * 3^2 * 5
        (3, 5, 9, 6, 7, 4, 8, 10)
        (1, 8, 2)(3, 4, 5)(6, 10, 7),
    Permutation group acting on a set of cardinality 10
    Order = 1
]
```

67.5 Representations of an Automorphism Group

To compute with automorphism groups, MAGMA uses various concrete representations of the group. These are summarised in this section.

`PermutationRepresentation(A)`

Construct a permutation representation of the group of automorphisms A. The function finds a union of conjugacy classes of the base group G which is closed under the action of A and with G-normal closure equal to G. The permutation action of A on such a set is faithful. The results returned are the representation of A as a homomorphism $A \to P$, the image of this homomorphism as a permutation group with standard support, and the set of elements of G used.

`PermutationGroup(A)`

Given a group of automorphisms A of a group G, this function returns a permutation group isomorphic to A as defined in the description of the function `PermutationRepresentation`.

`PermutationSupport(A)`

Given a group of automorphisms A of a group G, this function returns the set of elements of G (i.e., a union of conjugacy classes) used as the support of the permutation group constructed by the `PermutationRepresentation` function.

`PCGroupAutomorphismGroupPGroup(A)`

Attempt to directly construct a pc-representation for the group of automorphisms A of a conditioned p-group G. A must have been constructing using the `AutomorphismGroup` intrinsic, or any equivalent alias. The results returned are a boolean value indicating the solubility of A, and if soluble, a representation of A as a homomorphism $A \to P$ and the image of this homomorphism as a pc-group.

`FPGroup(A)`

A presentation for the group of automorphisms A on the generators of A. The isomorphism from the finitely presented group to the group of automorphisms A is also returned.

`OuterFPGroup(A)`

Suppose that A is the full group of automorphisms of a group G. This function returns a finitely presented group O isomorphic to the outer automorphism group of the base group G. The natural homomorphism from `FPGroup(A)` onto O is also returned.

Example H67E3

We calculate a permutation representation and presentation for the group of automorphisms of $PSL(2, 9)$.

```
> G := PGL(2, 9);
> A := AutomorphismGroup(G);
> PermutationGroup(A);
Permutation group acting on a set of cardinality 36
Order = 1440 = 2^5 * 3^2 * 5
    (1, 30)(3, 27)(5, 17)(6, 24)(8, 9)(10, 14)(11, 13)(12, 32) (15, 34)(16, 21)
       (18, 25)(19, 28)(22, 29)(23, 31)(26, 33)(35, 36)
    (1, 32, 19, 22)(2, 34)(3, 18, 7, 17)(4, 25, 30, 31) (5, 23, 33, 24)
       (6, 15, 26, 21)(8, 16, 29, 20)(9, 35, 14, 27) (10, 13, 11, 12)(28, 36)
    (1, 2, 3, 5, 8, 13, 22, 31)(4, 7, 9, 15, 24, 26, 34, 29)
       (6, 10, 17, 23, 32, 33, 28, 35)(11, 19, 27, 16, 25, 21, 30, 36)
       (12, 20, 14, 18)
    (1, 32, 33, 12, 21, 6)(2, 34, 26, 35, 17, 16)(3, 28, 22, 7, 18, 13)
       (4, 31, 20, 29, 24, 11)(5, 19, 25, 10, 15, 8)(9, 30, 36, 14, 23, 27)
> F<x, y, z, t> := FPGroup(A);
> F;
Finitely presented group F on 4 generators
Relations
    x^2 = Id(F)
    y^4 = Id(F)
    (x * y^-1)^5 = Id(F)
    y^-2 * x * y^-2 * x * y^-2 * x * y^2 * x * y^2 * x = Id(F)
    z^-1 * x * z * y^-1 * x^-1 * y^-2 * x^-1 * y * x^-1 *
       y^-2 * x^-1 * y * x^-1 * y^-1 * x^-1 = Id(F)
    z^-1 * y * z * y * x^-1 * y * x^-1 * y^-1 * x^-1 = Id(F)
    z^2 * y^-1 * x^-1 * y * x^-1 * y^-1 * x^-1 = Id(F)
    x^t = y * x * y^-1
    y^t = y^-1 * x * y * x * y
    z^t = z * x * y^-1 * x
    t^2 = x * y^2 * x * y^-1 * x * y
```

Example H67E4

We illustrate the process of finding a low degree permutation representation of an automorphism group using the above functions. We start with the Higman-Sims sporadic simple group, construct its automorphism group, and then use the function `PermutationGroup` to obtain a permutation representation.

```
> load hs100;
Loading "/home/magma/libs/pergps/hs100"
The simple group of Higman-Sims represented as a
permutation group of degree 100.
Order: 44 352 000 = 2^9 * 3^2 * 5^3 * 7 * 11.
Base: 1, 2, 3, 4, 5, 6.
```

```
Group: G
> aut := AutomorphismGroup(G);
> P := PermutationGroup(aut);
> P;
Permutation group P acting on a set of cardinality 5775
Order = 88704000 = 2^10 * 3^2 * 5^3 * 7 * 11
```

We've got a permutation representation on 5775 letters. Now we want to get it on 100 letters, so we need to find the subgroup of index 100.

```
> lix := LowIndexSubgroups(P, 100);
> [ Index(P, H) : H in lix];
[ 1, 2, 100 ]
```

There it is, so we can compute the corresponding permutation representation.

```
> H := CosetImage(P, lix[3]);
> H;
Permutation group H acting on a set of cardinality 100
> CompositionFactors(H);
    G
    |  Cyclic(2)
    *
    |  HS
    1
```

67.6 Automorphisms

The elements of a group of automorphisms are automorphisms of the base group, so MAGMA treats them as both homomorphisms and group elements. Thus they may be applied to elements and subgroups of the base group as a homomorphism, or they may be multiplied and have inverses taken as group elements. Of course, these last two operations are also homomorphism operations, being composition and the usual inverse of a bijection. Elements of a group of automorphisms are of type `GrpAutoElt`.

`A . i`

Let A be a group of automorphisms of a group G and let i be an integer such that $-n \leq i \leq n$, where n is the number of generators of A. This operator returns the i-th generator for A. A negative subscript indicates that the inverse of the generator is to be created. Finally, $A.0$ denotes the identity of A.

`Identity(A)`

`Id(A)`

`A ! 1`

The identity element of the group of automorphisms A.

`A ! f`

Let A be a group of automorphisms of a group G. Given an automorphism f of G, represented as a MAGMA map, this function returns the element of A corresponding to f. An error will result if f is not in the group generated by the generators of A. This uses the permutation representation of A to test for membership.

`Order(f)`

The order of the group automorphism f.

`f * g`

The product of the group automorphisms f and g. If f and g are regarded as maps, this function returns their composite: first apply f, then apply g.

`f ^ n`

The nth power of the group automorphism f. The integer n may be positive or negative.

`(g1, ..., gr)`

The left-normed commutator of the group automorphisms $g_1, \ldots, g_r$. Each of $g_1, \ldots, g_r$ must belong to a common automorphism group.

`g eq h`

Given group automorphisms g and h belonging to the same automorphism group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given group automorphisms g and h belonging to the same automorphism group, return `false` if g and h are the same element, `true` otherwise.

`IsInner(f)`

Returns `true` if the group automorphism f is an inner automorphism of the base group, `false` otherwise. If f is inner, then an element of the base group with conjugation action equal to the action of f is also returned.

Example H67E5

We illustrate some arithmetic operations with elements of the full group of automorphisms of a group of order 81.

```
> G := SmallGroup(81, 10);
> G;
GrpPC : G of order 81 = 3^4
PC-Relations:
  G.1^3 = G.4,
  G.2^3 = G.4^2,
  G.2^G.1 = G.2 * G.3,
  G.3^G.1 = G.3 * G.4
> A := AutomorphismGroup(G);
> #A;
486
> Ngens(A);
5
> IsInner(A.3);
false
> Order(A.3);
3
> A.3;
Automorphism of GrpPC : G of order 3 which maps:
  G.1 |--> G.1
  G.2 |--> G.2 * G.4^2
  G.3 |--> G.3
  G.4 |--> G.4
> A.3*A.4;
Automorphism of GrpPC : G which maps:
  G.1 |--> G.1
  G.2 |--> G.2 * G.3 * G.4^2
  G.3 |--> G.3 * G.4
  G.4 |--> G.4
> (A.3*A.4)^3;
Automorphism of GrpPC : G which maps:
  G.1 |--> G.1
  G.2 |--> G.2
  G.3 |--> G.3
  G.4 |--> G.4
> $1 eq Id(A);
true
```

Example H67E6

We can use the automorphism group machinery to determine the characteristic subgroups of a group.

```
> CharacteristicSubgroups := function(G)
```

```
>    local A, outers, NS, CS;
>    A := AutomorphismGroup(G);
>    outers := [ a : a in Generators(A) | not IsInner(a) ];
>    NS := NormalSubgroups(G);
>    CS := [n : n in NS | forall{a: a in outers| a(n`subgroup) eq n`subgroup }];
>    return CS;
> end function;
>
> CS := CharacteristicSubgroups(DirectProduct(Alt(4),Alt(4)));
> [c`order: c in CS];
[ 1, 16, 144 ]
> G := SmallGroup(512,298);
> #NormalSubgroups(G);
42
> #CharacteristicSubgroups(G);
28
```

67.7 Stored Attributes of an Automorphism Group

Groups of automorphisms have several attributes that may be stored as part of their data structure. The function `HasAttribute` is used to test if an attribute is stored and to retrieve its value, while the function `AssertAttribute` is used to set attribute values. The user is warned that when using `AssertAttribute` the data given is *not* checked for validity, apart from some simple type checks. *Setting attributes incorrectly will result in errors.*

`HasAttribute(A, s)`

`AssertAttribute(A, s, v)`

The `HasAttribute` function returns whether the group of automorphisms A has the attribute named by the string s defined and, if so, also returns the value of the attribute.

The `AssertAttribute` procedure sets the attribute of the group of automorphisms group named by string s to have value v. The possible names are:
`Group`: The base group of the automorphism group. This is always set.
`Order`: The order of the automorphism group. It is an integer and may be set by giving either an integer or a factored integer.
`OuterOrder`: The order of the outer automorphism group associated with A. It is an integer and may be set by giving either an integer or a factored integer.
`Soluble`: (also `Solvable`) A boolean value telling whether or not the automorphism group is soluble.
`InnerGenerators`: A sequence of generators of A known to be inner automorphisms.
`InnerMap`: A homomorphism from the base group to the automorphism group taking each base group element to its corresponding inner automorphism.
`ClassAction`: Stores the result of the `PermutationRepresentation` function call.

`ClassImage`: Stores the result of the `PermutationGroup` function call.
`ClassUnion`: Stores the result of the `ClassUnion` function call.
`FpGroup`: Stores the result of the `FPGroup` function call. The attribute is a pair $\langle F, m\rangle$ where F is an fp-group and m is an isomorphism $m : F \to A$. F and m are the two return values of `FPGroup(A)`.
`OuterFpGroup`: Stores the result of the `OuterFPGroup` function call. The attribute is a pair $\langle O, f\rangle$ where O is an fp-group and f is an epimorphism $f : F \to O$, where F is the first component of the `FpGroup` attribute. The kernel of f is the inner automorphism group. O and f are the two return values of `OuterFPGroup(A)`.
`GenWeights`: `WeightSubgroupOrders`: See the section on automorphism groups in the chapter on soluble groups for details.

Example H67E7

We select a group of order 904 from the small groups database and compute its group of automorphisms.

```
> G := SmallGroup(904, 4);
> FactoredOrder(G);
[ <2, 3>, <113, 1> ]
> FactoredOrder(Centre(G));
[ <2, 1> ]
> A := AutomorphismGroup(G);
> FactoredOrder(A);
[ <2, 7>, <7, 1>, <113, 1> ]
> HasAttribute(A, "FpGroup");
false
> HasAttribute(A, "OuterFpGroup");
false
> F,m := FPGroup(A);
> AssertAttribute(A, "FpGroup", <F,m>);
```

Note that values for some attributes, such as `FpGroup`, have not been calculated by the original `AutomorphismGroup` call, but they may be calculated and set later if desired. The outer automorphism group has order $2^5 \times 7$. We find the characteristic subgroups of `G`.

```
> n := NormalSubgroups(G);
> [x`order : x in n];
[ 1, 2, 113, 4, 226, 452, 452, 452, 904 ]
> characteristics := [s : x in n |
>   forall{f: f in Generators(A) | s@f eq s}
>   where s is x`subgroup];
> [#s : s in characteristics];
[ 1, 2, 113, 4, 226, 452, 904 ]
```

Note that two of the normal subgroups of order 452 are not characteristic.

Example H67E8

```
> G := AlternatingGroup(6);
> A := AutomorphismGroup(G);
> HasAttribute(A, "OuterFpGroup");
true
> F, f := FPGroup(A);
> O, g := OuterFPGroup(A);
> O;
Finitely presented group O on 2 generators
Relations
  O.1^2 = Id(O)
  O.2^2 = Id(O)
  (O.1 * O.2)^2 = Id(O)
> A`OuterOrder;
4
```

We find the outer automorphism group is elementary abelian of order 4. The direct product of `G` with itself has maximal subgroups isomorphic to `G`, in the form of diagonal subgroups. We can construct four non-conjugate examples using the outer automorphism group. The first example can be constructed without using an outer automorphism.

```
> GG, ins := DirectProduct(G, G);
> M := sub<GG|[(x@ins[1])*(x@ins[2]):x in Generators(G)]>;
> IsMaximal(GG, M);
true
```

The subgroup `M` is the first, the obvious diagonal, constructed using just the embeddings returned by the direct product function. We get others by twisting the second factor using an outer automorphism. First we get (representatives of) the outer automorphisms explicitly.

```
> outers := {x @@ g @ f : x in [O.1, O.2, O.1*O.2]};
> Representative(outers);
Automorphism of GrpPerm: G, Degree 6, Order 2^3 * 3^2 * 5 which maps:
  (1, 2)(3, 4, 5, 6) |--> (1, 3, 6, 2)(4, 5)
  (1, 2, 3) |--> (1, 4, 2)(3, 5, 6)
```

The set `outers` now contains three distinct outer automorphisms of `G`. We use them to get three more diagonal subgroups.

```
> diags := [M] cat
> [sub<GG|[(x @ ins[1])*(x @ f @ ins[2]):x in Generators(G)]>:
>   f in outers];
> [IsMaximal(GG, m) : m in diags];
[ true, true, true, true ]
> IsConjugate(GG, diags[2], diags[4]);
false
```

The other five tests for conjugacy will give similarly negative results.

67.8 Holomorphs

Given a group G and the full group of automorphisms A of G then the holomorph of G is the semidirect product $G \times_\theta A$, where $\theta : A \to Aut(G)$ is the identity map.

`Holomorph(G)`

`Holomorph(GrpFP, G)`

Given a finite permutation, matrix or pc-group G with full group of automorphisms A, this function returns the semidirect product E of G by A. The group E is returned as a permutation group (or a finitely presented group if `GrpFP` is specified) of degree $|G|$ in which G is a regular normal subgroup, and A is the stabilizer of the point 1. The embedding map $G \to E$, and the natural epimorphism $E \to A$ are also returned. In the returned group E, the generators of G appear first, followed by those of A.

`Holomorph(G, A)`

`Holomorph(GrpFP, G, A)`

Given a finite permutation, matrix or pc-group G and a group of automorphisms A, this function returns the semidirect product E of G by A. The group E is returned as a permutation group (or a finitely presented group if `GrpFP` is specified) of degree $|G|$ in which G is a regular normal subgroup, and A is the stabilizer of the point 1. The embedding map $G \to E$, and the natural epimorphism $E \to A$ are also returned. In the returned group E, the generators of G appear first, followed by those of A.

Example H67E9

We construct the holomorph of the group $G = PGL(2, 9)$.

```
> G := PGL(2, 9);
> E := Holomorph(G); E;
Permutation group E acting on a set of cardinality 720
> #E;
1036800
> CompositionFactors(E);
    G
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Alternating(6)
    *
    |  Alternating(6)
    1
```

67.9 Bibliography

[**CH03**] J.J. Cannon and D.F. Holt. Automorphism group computation and isomorphism testing in finite groups. *J. Symbolic Comp.*, 35(3):241–267, 2003.

[**ELGO02**] Bettina Eick, C.R. Leedham-Green, and E.A. O'Brien. Constructing automorphism groups of a p-groups. *Comm. Algebra*, 30:2271–2295, 2002.

[**Smi94**] Michael J. Smith. *Computing automorphisms of finite soluble groups.* PhD thesis, Australian National University, 1994.

68 COHOMOLOGY AND EXTENSIONS

Chapter 68
COHOMOLOGY AND EXTENSIONS

68.1 Introduction

The following collection of cohomology functions is designed to provide a flexible set of tools for computing with first and second cohomology groups of any type of finite group acting on any reasonable module, including a module defined by an action on an arbitrary finitely generated abelian group. First (but not second) cohomology groups can also be calculated for infinite groups defined by a finite presentation.

Zero-cocycles, one-cocycles and two-cocycles may be computed and identified. Extensions of modules by groups can be constructed as finitely presented groups, or as PC-groups when the acting group is a PC-group. It is also possible to compute a representative set of extensions of the module by the group each of which is distinct up to a group isomorphism fixing the module. These functions complement, but do not completely supplant, an older collection of functions pertaining to cohomology groups, Schur multiplicators and covering groups which apply to permutation groups (see Chapter 58 on Permutation Groups).

The first cohomology group $H^1(G, M)$ is calculated as the nullspace of a certain matrix. The details can be found in Section 5 of [CCH01]. This immediately allows manipulation and identification of one-cocycles. The second cohomology group $H^2(G, M)$ is more difficult to compute. While it can also be found as the nullspace of a suitable matrix, this matrix can be uncomfortably large in big examples. For soluble groups defined by a PC-presentation, the matrix corresponds to solving the consistency equations for a PC-presentation of a general extension of the module by the group, which depends on the number of group generators rather than its order, and is manageable for quite large groups. For permutation and matrix groups G, the size of the matrix for which the nullspace is required is much larger, but can often be reduced to a reasonable size by using a base and strong generating set for G. In the case where only the dimension of $H^2(G, M)$ is required, and M is a module over a finite field of prime order p, then the calculation of this dimension can be reduced to the determination of $H^2(Q, M)$ for a suitable collection of p-subgroups Q of G. The latter calculation can be carried out efficiently using the PC-presentation approach (see [Hol85b] for details).

To use the new functions, the user must initially invoke the function `CohomologyModule`, which creates a special object for the group action corresponding to the module, and all subsequent (new) cohomology functions take this object as their first argument.

In the case of a finite permutation or matrix group G acting on a module M over a prime field, the dimension of $H^2(G, M)$ may be found much more quickly by executing `CohomologicalDimension(CM, 2)`, where CM is the cohomology module for the action of G on M, rather than by invoking `Dimension(CohomologyGroup(CM, 2))`. However, the

former call does not allow the possibility of subsequent calculations with two-cocycles or extensions.

The equivalent older function, `CohomologicalDimension(G, M, 2);` (for a permutation group G) is often faster still for small examples, but the new function will succeed on much larger examples than the old. For the convenience of the reader, some of these older functions are described in this section of the Handbook. For complete details about the older functions, see the section on cohomology in the chapter on Permutation Groups.

68.2 Creation of a Cohomology Module

In order to compute the cohomology of a group with respect to a G-module M, it is first necessary to construct a data structure known as a *cohomology module*.

`CohomologyModule(G, M)`

Given a group G and a G-module M with acting group G this function returns a cohomology module for the action of G. The group G may be a finite permutation group, a finite matrix group, a PC-group, or any finitely presented group. For the PC-group case, however, the PC-presentation of G must be conditioned. This can be achieved by first executing the statement `G := ConditionedGroup(G);`

`CohomologyModule(G, Q, T)`

Let G be a group which acts on a finitely-generated abelian group with invariants given by the sequence Q, and action described by T. The action T is given in the form of a sequence of $d \times d$ matrices over the integers, where d is the length of T, and `T[i]` defines the action of the i-th generator of G on the abelian group. The function returns a cohomology module for the action of G. The group G may be a finite permutation group, a finite matrix group, a PC-group or any finitely presented group. For the PC-group case, however, the PC-presentation of G must be conditioned. This can be achieved by first executing the statement `G := ConditionedGroup(G);`

Example H68E1

We construct the cohomology module for $PSL(3,2)$ acting on a module of dimension 3 over $GF(2)$. We first need to find a module of dimension 3.

```
> G := PSL(3, 2);
> Irrs := AbsolutelyIrreducibleModules(G, GF(2));
> Irrs;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 3 over GF(2),
    GModule of dimension 3 over GF(2),
    GModule of dimension 8 over GF(2)
]
> M := Irrs[2];
> CM := CohomologyModule(G, M);
```

```
> CM;
Cohomology Module
```

`CohomologyModule(G, A, M)`

For a permutation group G acting on some abelian group A through M, compute the cohomology module. M has to be either a map from G into the endomorphisms of A, or a sequence of endomorphisms of A, one for each of the generators of G.

68.3 Accessing Properties of the Cohomology Module

The functions described in this section merely return data used to define the cohomology module. In each case, the argument CM must be a cohomology module returned by a call to `CohomologyModule`.

`Module(CM)`

The $K[G]$-module used to define the cohomology module CM. An error occurs if CM was defined by an action on a finitely generated abelian group.

`Invariants(CM)`

Given a cohomology module CM that was defined by an action on a finitely generated abelian group A, return the invariants of A. If CM was not defined by an action on an abelian group, an error results.

`Dimension(CM)`

Let CM be a cohomology module. If CM was defined by the action of a group on an R-module M, return the dimension of M. In the case in which CM was defined by the action of a group on a finitely generated abelian group A, the rank of A is returned.

`Ring(CM)`

The ring over which the module used to define the cohomology module CM is defined. If CM is defined in terms of an action on a finitely generated abelian group A, then the ring will be the integers if A is infinite, and the integers modulo the exponent of A if A is finite.

`Group(CM)`

The group used to define action on the cohomology module CM.

`FPGroup(CM)`

Given a cohomology module CM with associated group G, return a finitely presented group F isomorphic to G and the isomorphism from F to G. This presentation is on a strong generating set if G is a permutation or matrix group. It is used in the construction of presentations of extensions returned by the function `Extension`.

`MatrixOfElement(CM, g)`

The matrix representing the action of the element g in the group of CM on the module of CM.

68.4 Calculating Cohomology

`CohomologyGroup(CM, n)`

Given a cohomology module CM for the group G acting on the module M and a non-negative integer n taking one of the values 0, 1 or 2, this function returns the cohomology group $H^n(G, M)$. For modules defined over the ring of integers only, n may also be equal to 3. (In this case, $H^3(G, M)$ is computed as the second cohomology group of M regarded as a module over Q/Z.) If the group used to define CM was a finitely presented group, then n may only be equal to 0 or 1. Note that CM must be a module returned by invoking `CohomologyModule`.

`CohomologicalDimension(CM, n)`

Given a cohomology module CM for the group G acting on the module M defined over a finite field K and a non-negative integer n taking one of the values 0, 1 or 2, this function returns the dimension of $H^n(G, M)$ over K. Note that this function may only be applied to the module returned by a call to `CohomologyModule(G, M)`, where M is a module over a finite field K. When $n = 2$, this function is faster and may be applied to much larger examples than `CohomologyGroup(CM, n)` but, unlike that function, it does not enable the user to compute with explicit extensions and two-cocycles.

Note that there are some alternative functions for performing these calculations described in other manual chapters.

`CohomologicalDimension(M, n)`

For $K[G]$-module M (with K a finite field and G a finite group), compute and return the K-dimension of the cohomology group $H^n(G, M)$ for $n \geq 0$. For $n = 0$ and 1, this is carried out by using the function `CohomologicalDimension(CM,n)` just described. For $n \geq 2$, it is done recursively using projective covers and dimension shifting to reduce to the case $n = 1$.

`CohomologicalDimensions(M, n)`

For $K[G]$-module M (with K a finite field and G a finite group), compute and return the sequence of K-dimensions of the cohomology groups $H^k(G, M)$ for $1 \leq k \leq n$. On account of the recursive method used, this is quicker than computing them all individually.

`CohomologicalDimension(G, M, n)`

Given the permutation group G, the $K[G]$-module M and an integer n (equal to 1 or 2), return the dimension of the n-th cohomology group of G acting on M. *Note that K must be a finite field of prime order.* This function invokes Derek Holt's original C cohomology code (see [Hol85b]). In some cases it will be faster than the function that uses the cohomology module data structure.

Example H68E2

We examine the first and second cohomology groups of the group A_8.

```
> G := Alt(8);
> M := PermutationModule(G, GF(3));
```

We first calculate the dimensions of $H^1(G, M)$ and $H^2(G, M)$ using the old functions.

```
> time CohomologicalDimension(G, M, 1);
0
Time: 0.020
> time CohomologicalDimension(G, M, 2);
1
Time: 0.020
```

We now recalculate the dimensions of $H^1(G, M)$ and $H^2(G, M)$ using the new functions.

```
> X := CohomologyModule(G, M);
> time CohomologicalDimension(X, 1);
0
Time: 0.020
> time CohomologicalDimension(X, 2);
1
Time: 0.920
> X := CohomologyModule(G, M);
> time C:=CohomologyGroup(X, 2);
Time: 4.070
> C;
Full Vector space of degree 1 over GF(3)
```

Example H68E3

In the case of $\Omega^-(8, 3)$ acting on its natural module, the new function succeeds, but the old function does not.

```
> G := OmegaMinus(8, 3);
> M := GModule(G);
> X := CohomologyModule(G, M);
> time CohomologicalDimension(X, 2);
2
Time: 290.280
> phi, P := PermutationRepresentation(G);
```

```
> MM := GModule(P, [ActionGenerator(M, i): i in [1..Ngens(G)]] );
>  time CohomologicalDimension(P, MM, 2);
Out of space.
>> time CohomologicalDimension(P, MM, 2);
                                ^
Runtime error in 'CohomologicalDimension': Cohomology failed
```

68.5 Cocycles

Before invoking the functions in this section, it is necessary to first invoke the function `CohomologyGroup(CM, n)` for the appropriate n.

For $n = 0, 1$ or 2, an *n-cocycle* is a function from G^n to the module M, where elements of G^n are represented as an n-tuple $\langle g_1, \ldots, g_n \rangle$ of group elements, for which a certain relation is satisfied. These relations are consistent with the MAGMA convention of the use of right actions, and so they are slightly different from those encountered in many textbooks, where left actions are more common.

0-, 1- and 2-cocycles z, o and t, respectively, satisfy the following relations for all $g, h, \in G$.

$$z(\langle\rangle)^g = z(\langle\rangle);$$

$$o(\langle gh \rangle) = o(\langle g \rangle)^h + o(\langle h \rangle);$$

$$t(\langle gh, k \rangle) + t(\langle g, h \rangle)^k = t(\langle g, hk \rangle) + t(\langle h, k \rangle).$$

`ZeroCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and an element s of the cohomology group $H^0(G, M)$ associated with CM, this function returns the corresponding zero-cocycle. The zero-cocycle is returned as a function of the 0-tuple $\langle\rangle$, of which the image is an element of the fixed point submodule of M. The argument s may either be given as an element of $H^0(G, M)$ or as a sequence of integers defining such an element.

`IdentifyZeroCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and a zero-cocycle given as a function of the 0-tuple $\langle\rangle$, of which the image is an element of the fixed point submodule of M, this function returns the corresponding element of $H^0(G, M)$. Hence this function is the inverse function to `ZeroCocycle`.

`OneCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and an element s of the cohomology group $H^1(G, M)$ associated with CM, the function returns a corresponding one-cocycle. The one-cocycle is returned as a function from G to the module M, where elements g of G are represented as 1-tuples $\langle g \rangle$. The argument s may either be given as an element of $H^1(G, M)$ or as a sequence of integers defining such an element.

`IdentifyOneCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and a one-cycle s for CM, specified as a function from G to the module M (where elements g of G are represented as 1-tuples $\langle g \rangle$), this function returns the corresponding element of $H^1(G, M)$. Thus, the function is the inverse to `OneCocycle`.

`IsOneCoboundary(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and a one-cycle s for CM, specified as a function from G to the module M (where elements g of G are represented as 1-tuples $\langle g \rangle$), this function determines whether the cocyle is a 1-coboundary; that is, whether it corresponds to the zero element of $H^1(G, M)$. If so, then it also returns a corresponding 0-cochain $t(\langle\rangle)$ that satisfies $s(\langle g \rangle) = t(\langle\rangle) - t(\langle\rangle)^g$ for all $g \in G$.

`TwoCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and an element s of the cohomology group $H^2(G, M)$ associated with CM, the function returns a corresponding two-cocycle. The two-cocycle is returned as a function from $G \times G$ to the module M, where elements of $G \times G$ are represented as 2-tuples $\langle g_1, g_2 \rangle$. The argument s may either be given as an element of $H^2(G, M)$ or as a sequence of integers defining such an element.

`IdentifyTwoCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and a two-cycle s for CM, specified as a function from $G \times G$ to the module M (where elements of $G \times G$ are represented as 2-tuples $\langle g_1, g_2 \rangle$), this function returns the corresponding element of $H^2(G, M)$. Thus, the function is the inverse to `TwoCocycle`.

`IsTwoCoboundary(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and a two-cycle s for CM, specified as a function from G to the module M (where elements of $G \times G$ are represented as 2-tuples $\langle g_1, g_2 \rangle$), this function determines whether the cocyle is a 2-coboundary; that is, whether it corresponds to the zero element of $H^2(G, M)$. If so, then it also returns a corresponding 1-cochain $t(\langle g \rangle)$ that satisfies $s(\langle g, h \rangle) = t(\langle g \rangle)^h + t(\langle h \rangle) - t(\langle gh \rangle)$ for all $g, h \in G$.

Example H68E4

An easy example where the module is an abelian group defined by its invariant factors.

```
> G := PermutationGroup< 4 | (1,2,3,4) >;
> invar:=[2,4,4];
> mats := [ Matrix(Integers(),3,3,[1,2,0,0,0,1,0,1,2]) ];
> X := CohomologyModule(G,invar,mats);
> C := CohomologyGroup(X,0);
> C;
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 4 ]
> ZeroCocycle(X,[3]);
function(tp) ... end function
> IdentifyZeroCocycle(X,func<x|-$1(<>)>);
(1)
> C := CohomologyGroup(X,1);
> C;
Full Quotient RSpace of degree 2 over Integer Ring
Column moduli:
[ 2, 2 ]
> z1 := OneCocycle(X,[1,0]);
> z2 := OneCocycle(X,[0,1]);
> z1(<G.1>);
(1 0 0)
> z := func< x | z1(x)+z2(x) >;
> IdentifyOneCocycle(X,z);
(1 1)
> C := CohomologyGroup(X,2);
> C;
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 4 ]
> z1 := TwoCocycle(X,[1]);
> z1(<G.1,G.1^2>);
(1 1 3)
> z := func< xy | z1(xy)+z1(xy) >;
> IdentifyTwoCocycle(X,z);
(2)
```

68.6 The Restriction to a Subgroup

`Restriction(CM, H)`

Given a cohomology module for a group G and a subgroup H of G, form the restriction of the input cohomology module to H.

Note that, denoting this restriction by CMH, we can define the restriction maps on the first and second cohomology groups of CM by

```
>  res1 := hom<CohomologyGroup(CM, 1) -> CohomologyGroup(CMH, 1) |
>                          x:->IdentifyOneCocycle(CMH,OneCocycle(CM,x)) >;
>  res2 := hom<CohomologyGroup(CM, 2) -> CohomologyGroup(CMH, 2) |
>                          x:->IdentifyTwoCocycle(CMH,TwoCocycle(CM,x)) >;
```

Example H68E5

In this example we define G to be the group $GL(3,2)$ and H to be the Sylow 2-subgroup of G. We illustrate how to calculate the restriction mappings of $H^n(G,M)$ to $H^n(G,MH)$, where MH is the restriction of M to H.

```
> G := GL(3, 2);
> M := GModule(G);
> H := Sylow(G, 2);
> CG := CohomologyModule(G, M);
> CH := Restriction(CG, H);
```

We first consider $H^1(G,M)$.

```
> H1G := CohomologyGroup(CG, 1); H1G;
Full Vector space of degree 1 over GF(2)
> H1H := CohomologyGroup(CH, 1); H1H;
Full Vector space of degree 2 over GF(2)
> res1 := hom<H1G -> H1H | x:->IdentifyOneCocycle(CH,OneCocycle(CG,x)) >;
> res1(H1G.1);
(1 1)
```

We now consider $H^2(G,M)$.

```
> H2G := CohomologyGroup(CG, 2); H2G;
Full Vector space of degree 1 over GF(2)
> H2H := CohomologyGroup(CH, 2); H2H;
Full Vector space of degree 3 over GF(2)
> res2 := hom<H2G -> H2H | x:-> IdentifyTwoCocycle(CH,TwoCocycle(CG,x)) >;
> res2(H2G.1);
(0 0 1)
```

In the case of a zero restriction, we can find a corresponding coboundary.

```
> H:=sub< G | G.2, G.2^(G.1*G.2*G.1) >;
> #H;
21
```

```
> CH := Restriction(CG, H);
> CohomologyGroup(CH, 1); CohomologyGroup(CH, 2);
Full Vector space of degree 0 over GF(2)
Full Vector space of degree 0 over GF(2)
> t:=TwoCocycle(CG,[1]);
> isc, o := IsTwoCoboundary(CH, t);
> isc;
true
> forall{ <h,k> : h in H, k in H | t(<h,k>) eq
>            o(<h>)*MatrixOfElement(CH,k) + o(<k>) - o(<h*k>) };
true
```

68.7 Other Operations on Cohomology Modules

`CorestrictionMapImage(G, C, c, i)`

`CorestrictCocycle(G, C, c, i)`

Given an i-cochain c for the cohomology module C which has to be defined wrt. to some subgroup U of G, return the corestriction of c to $H^i(G, \ldots)$.

`InflationMapImage(M, c)`

`LiftCocycle(M, c)`

`NewCodomain`	ANY	*Default :* `false`
`Level`	RNGINTELT	*Default :* `false`

Given a cochain $c : G^i \to X$ and a (transversal) map $H \to G$, return the inflation (lift) of c to H, ie. a cochain $d : H^i \to X$ defined by $d(h) := c(M(h))$. If `Level` is given c is assumed to be in the cohomology group of that level, ie. i :=`Level`. If `Level` is not specified, MAGMA tries its best to guess the correct level.

If `NewCodomain` is given, the values of d are coerced into this structure.

`CoboundaryMapImage(M, i, c)`

For a cohomology module M, a level i and a i-cochain c (as a user program), return a $i+1$-coboundary as obtained from the cohomological coboundary operator.

68.8 Constructing Extensions

`Extension(CM, s)`

Given the cohomology module CM for the group G acting on the module M and an element s of $H^2(G, M)$, this function returns the corresponding extension E of the module M by G as a finitely presented group. The generators of E are chosen so that the generators of the acting group G (or rather strong generators for G when G is a permutation or matrix group) come first, and the generators of M come last. The argument s should be either an element of $H^2(G, M)$ or a sequence of integers defining such an element.

The projection from E to G and the injection from an abelian group isomorphic to M to E are also returned.

This function may only be applied when the module M used to define CM is defined over a finite field of prime order, the integers, or as an abelian group in a call of `CohomologyModule(G, Q, T)`.

`SplitExtension(CM)`

Given the cohomology module CM for the group G acting on the module M, this function returns the split extension E of the module M by G as a finitely presented group. The generators of E are chosen so that the generators of the acting group G (or strong generators for G when G is a permutation or matrix group) come first, and the generators of M come last. The extension returned is the same as for `Extension(CM, s)` with s taken as the zero element of $H^2(G, M)$, but `SplitExtension` is much faster, and does not require $H^2(G, M)$ to be calculated first. This function will also work when the group used to define CM was a finitely presented group.

The projection from E to G and the injection from an abelian group isomorphic to M to E are also returned.

This function may only be applied when the module M used to define CM is defined over a finite field of prime order, the integers, or as an abelian group in a call of `CohomologyModule(G, Q, T)`.

`pMultiplicator(G, p)`

Given the permutation group G and a prime p dividing the order of G, return the invariant factors of the p-part of the Schur multiplicator of G. This function calls Derek Holt's original cohomology code (see [Hol84]).

`pCover(G, F, p)`

Given the permutation group G and the finitely presented group F such that G is an epimorphic image of F in the sense described below, and a prime p, return a presentation for the p-cover of G, constructed as an extension of the p-multiplier by F. Note that the epimorphism of F onto G must satisfy the conditions that, firstly, the generators of F are in one-to-one correspondence with those of G and, secondly, the relations of F are satisfied by the generators of G. In other words, the mapping

taking the i-th generator of F to the i-th generator of G must be an epimorphism. Usually this mapping will be an isomorphism, although this is not mandatory. This function calls Derek Holt's original cohomology code (see [Hol85a]).

Example H68E6

We apply the machinery to construct a non-split extension of the elementary abelian group of order 3^8 by A_8.

```
> G := Alt(8);
> M := PermutationModule(G,GF(3));
> X := CohomologyModule(G,M);
> C := CohomologyGroup(X,2);
> C;
Full Vector space of degree 1 over GF(3)
```

The function `Extension` is used to construct a non-split extension E of the module M by the group G.

```
> E := Extension(X,[1]);
```

The object E is a finitely presented group, in which the 8 module generators come last. We now construct a (rather large-degree) faithful permutation representation of E.

```
> n := Ngens(E);
> D := sub< E | [E.i : i in [n-7..n-1]] >;
> ct := CosetTable(E,D:CosetLimit:=10^6,Hard:=true);
> P := CosetTableToPermutationGroup(E,ct);
> Degree(P);
60480
> #P eq 3^8 * #G;
true
```

We extract the normal subgroup of order 3^8 of the extension E, and verify that the extension is non-split.

```
> Q := sub<P | [P.i : i in [n-7..n]] >;
> #Q eq 3^8;
true
> IsNormal(P,Q);
true
> Complements(P,Q);
[]
```

Example H68E7

We investigate the cohomology of the permutation module for A_5 taken over the integers.

```
> G := Alt(5);
> M := PermutationModule(G,Integers());
> X := CohomologyModule(G,M);
```

```
> CohomologyGroup(X,0);
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 0 ]
> CohomologyGroup(X,1);
Full Quotient RSpace of degree 0 over Integer Ring
Column moduli:
[ ]
> CohomologyGroup(X,2);
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 3 ]
```

While we can form extensions of M in this case, we are unable to determine the distinct extensions.

```
> E := Extension(X,[1]);
> E;
Finitely presented group E on 8 generators
Relations
    (E.4, E.5) = Id(E)
    (E.4, E.6) = Id(E)
    (E.4, E.7) = Id(E)
    (E.4, E.8) = Id(E)
    (E.5, E.6) = Id(E)
    (E.5, E.7) = Id(E)
    (E.5, E.8) = Id(E)
    (E.6, E.7) = Id(E)
    (E.6, E.8) = Id(E)
    (E.7, E.8) = Id(E)
    (E.1, E.4^-1) = Id(E)
    (E.1, E.5^-1) = Id(E)
    E.1^-1 * E.6 * E.1 * E.7^-1 = Id(E)
    E.1^-1 * E.7 * E.1 * E.8^-1 = Id(E)
    E.1^-1 * E.8 * E.1 * E.6^-1 = Id(E)
    E.2^-1 * E.4 * E.2 * E.5^-1 = Id(E)
    E.2^-1 * E.5 * E.2 * E.6^-1 = Id(E)
    E.2^-1 * E.6 * E.2 * E.4^-1 = Id(E)
    (E.2, E.7^-1) = Id(E)
    (E.2, E.8^-1) = Id(E)
    (E.3, E.4^-1) = Id(E)
    E.3^-1 * E.5 * E.3 * E.6^-1 = Id(E)
    E.3^-1 * E.6 * E.3 * E.7^-1 = Id(E)
    E.3^-1 * E.7 * E.3 * E.5^-1 = Id(E)
    (E.3, E.8^-1) = Id(E)
    E.1^-3 * E.4^-1 * E.5^-2 = Id(E)
    (E.1^-1 * E.3^-1)^2 = Id(E)
    E.3^-3 * E.4 * E.8^2 = Id(E)
    E.2^-1 * E.1 * E.3^-1 * E.2 * E.1^-1 * E.4^-1 * E.8^2 = Id(E)
    E.2 * E.3 * E.2 * E.3 * E.8^-4 = Id(E)
```

```
    E.2^-1 * E.3^-1 * E.2^2 * E.3^-1 * E.4 * E.5 * E.6^-2 * E.7 = Id(E)
> DE := DistinctExtensions(X);
Sorry, can only compute distinct extensions over prime field or finite abelian
group
```

68.9 Constructing Distinct Extensions

The functions below compute the distinct extensions of one group by another.

`DistinctExtensions(CM)`

Given the cohomology module CM for the group G acting on the module M, this function returns a sequence containing all of the distinct extensions of the module M by G, each in the form returned by `Extension(CM, s)`. Two such extensions E_1, E_2 are regarded as being distinct if there is no group isomorphism from one to the other that maps the subgroup of E_1 corresponding to M to the subgroup of E_2 corresponding to M.

This function may only be applied when the module M used to define CM is defined over a finite field of prime order, the integers, or as an abelian group in a call of `CohomologyModule(G, Q, T)`.

Example H68E8

We consider the extensions of the trivial module over $GF(2)$ by the group $Z_2 \times Z_2$.

```
> G := DirectProduct(CyclicGroup(2),CyclicGroup(2));
> M := TrivialModule(G,GF(2));
> C := CohomologyModule(G,M);
> CohomologicalDimension(C,2);
3
> D := DistinctExtensions(C);
> #D;
4
```

So there are $2^3 = 8$ equivalence classes of extensions. But only four are distinct up to an isomorphism fixing the module. To examine them, we form permutation representations:

```
> DP := [ CosetImage(g,sub<g|>) : g in D ];
> [IsAbelian(d): d in DP];
[ true, true, false, false ]
// the first two are abelian
> [IsIsomorphic(d,DihedralGroup(4)) : d in DP];
[ false, false, true, false ]
// The third one is dihedral
> #[g : g in DP[4] | Order(g) eq 4];
6
```

So the fourth group must be the quaternion group.

`ExtensionsOfElementaryAbelianGroup(p, d, G)`

Given a prime p, a positive integer d, and a permutation group G, this function returns a list of finitely presented groups which are isomorphic to the distinct extensions of an elementary abelian group N of order p^d by G. Two such extensions E_1 and E_2 with normal subgroups N_1 and N_2 isomorphic to N are considered to be distinct if there is no group isomorphism $G_1 \to G_2$ that maps N_1 to N_2. Each extension E is defined on $d+r$ generators, where r is the number of generators of G. The last d of these generators generate the normal subgroup N, and the quotient of E by N is a presentation of G on its own generators.

Example H68E9

We form the distinct extensions of the elementary abelian group $Z_2 \times Z_2$ by the alternating group A_4.

```
> E := ExtensionsOfElementaryAbelianGroup(2,2,Alt(4));
> #E;
4
```

So there are four distinct extensions of an elementary group of order 4 by A_4

```
> EP := [ CosetImage(g,sub<g|>) : g in E ];
> [#Centre(e): e in EP];
[ 1, 1, 4, 4 ]
```

The first two have nontrivial action on the module. The module generators in the extensions come last, so these will be $e.3$ and $e.4$. We can use this to test which of the extensions are non-split.

```
> [ Complements(e,sub<e|e.3,e.4>) eq [] : e in EP];
[ false, true, false, true ]
> AbelianInvariants(Sylow(EP[2],2));
[ 4, 4 ]
```

So the first and fourth extensions split and the second and third do not. $EP[2]$ has a normal abelian subgroup of type $[4, 4]$.

`ExtensionsOfSolubleGroup(H, G)`

Given permutation groups G and H, where H is soluble, this function returns a sequence of finitely presented groups, the terms of which are isomorphic to the distinct extensions of H by G. Two such extensions E_1 and E_2 with normal subgroups H_1 and H_2 isomorphic to H are considered to be distinct if there is no group isomorphism $G_1 \to G_2$ that maps H_1 to H_2. Each extension E is defined on $d + r$ generators, where the last d generators generate the normal subgroup H, and the quotient of E by H is a presentation for G on its own generators. (The last d generators of E do not correspond to the original generators of H, but to a PC-generating sequence for H.)

Example H68E10

How many extensions are there of a dihedral group of order 8 by itself? This calculation is currently rather slow.

```
> D4 := DihedralGroup(4);
> time S := ExtensionsOfSolubleGroup(D4, D4);
Time: 120.210
> #S;
20
> ES := [CosetImage(g,sub<g|>) : g in S ];
> [#Centre(g): g in ES];
[ 4, 2, 4, 2, 4, 2, 2, 4, 2, 4, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2 ]
> [NilpotencyClass(g) : g in ES ];
[ 2, 3, 2, 3, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
> [Exponent(g): g in ES];
[ 4, 8, 4, 8, 4, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8 ]
```

Example H68E11

We determine the distinct extensions of the abelian group with invariants $[2, 4, 4]$ by the cyclic group of order 4.

```
> Z := Integers();
> G := PermutationGroup<4 | (1,2,4,3)>;
> Q := [2, 4, 4];
> T := [ Matrix(Z,3,3,[1,2,0,0,0,1,0,1,2]) ];
> CM := CohomologyModule(G, Q, T);
> extns := DistinctExtensions(CM);
> extns;
[
    Finitely presented group on 4 generators
    Relations
        $.2^2 = Id($)
        $.3^4 = Id($)
        $.4^4 = Id($)
        ($.2, $.3) = Id($)
        ($.2, $.4) = Id($)
        ($.3, $.4) = Id($)
        $.1^-1 * $.2 * $.1 * $.3^-2 * $.2^-1 = Id($)
        $.1^-1 * $.3 * $.1 * $.4^-1 = Id($)
        $.1^-1 * $.4 * $.1 * $.4^-2 * $.3^-1 = Id($)
        $.1^4 = Id($),
    Finitely presented group on 4 generators
    Relations
        $.2^2 = Id($)
        $.3^4 = Id($)
        $.4^4 = Id($)
        ($.2, $.3) = Id($)
```

```
        ($.2, $.4) = Id($)
        ($.3, $.4) = Id($)
        $.1^-1 * $.2 * $.1 * $.3^-2 * $.2^-1 = Id($)
        $.1^-1 * $.3 * $.1 * $.4^-1 = Id($)
        $.1^-1 * $.4 * $.1 * $.4^-2 * $.3^-1 = Id($)
        $.1^4 * $.2^-1 * $.3^-1 * $.4^-3 = Id($),
    Finitely presented group on 4 generators
    Relations
        $.2^2 = Id($)
        $.3^4 = Id($)
        $.4^4 = Id($)
        ($.2, $.3) = Id($)
        ($.2, $.4) = Id($)
        ($.3, $.4) = Id($)
        $.1^-1 * $.2 * $.1 * $.3^-2 * $.2^-1 = Id($)
        $.1^-1 * $.3 * $.1 * $.4^-1 = Id($)
        $.1^-1 * $.4 * $.1 * $.4^-2 * $.3^-1 = Id($)
        $.1^4 * $.3^-2 * $.4^-2 = Id($)
]
```

Since the extensions are soluble groups, we construct pc-presentations of each and verify that no two of the groups are isomorphic.

```
> E1 := SolubleQuotient(extns[1]);
> E2 := SolubleQuotient(extns[2]);
> E3 := SolubleQuotient(extns[3]);
> IsIsomorphic(E1, E2);
false
> IsIsomorphic(E1, E3);
false
> IsIsomorphic(E2, E3);
false
```

`IsExtensionOf(G)`

`Degree`	RngInt	*Default :* 0
`MaxId`	RngInt	*Default :* 15
`DegreeBound`	RngInt	*Default :* ∞

For a given permutation group G, find normal abelian subgroup $A < G$ such that G can be obtained by extending G/A by A. The function returns a sequence of tuples T containing

- the cohomology module of G/A acting on A
- the 2-cocyle as an element in $H^2(G/A, A)$ corresponding to G
- the actual 2-cocyle as a user defined function

- a pair $\langle a, b\rangle$ giving the degree a of the transitive group G/A and the number b identifying the group in the data base. If b is larger than 20 (or `MaxId`) the hash value of the group is returned instead.
- the abelian invariants of A
- a set containing all pairs $\langle a, b\rangle$ such that ${}_aT_b$ can be obtained through this extension process.

If `DegreeBound` is given, only subgroups A are considered such that G/A has less than `DegreeBound` many elements. The list considered contains only sugroups that are maximal under the restrictions. If `Degree` is given, G/A must have exactly `Degree` many elements.

`IsExtensionOf(L)`

`Degree`	RngInt	*Default :* 0
`MaxId`	RngInt	*Default :* 15
`DegreeBound`	RngInt	*Default :* ∞

For all groups G in L, `IsExtensionOf` is called. The first sequence returned contains tuples as in `IsExtensionOf` above. The sequence is minimal such that all groups in L can be generated using the cohomology modules in the sequence. The second return value contains a set of pairs $\langle a, b\rangle$ describing all transitive groups that can be obtained through the processes.

68.10 Finite Group Cohomology

This section describes Magma functions for computing the first cohomology group of a finite group with coefficient in a finite (not necessarily abelian) group. These functions are based on [Hal05].

Let Γ be a group. A group A on which Γ acts by group automorphisms from the right, is called a Γ-*group*. Given a Γ-group A, define

$$H^0(\Gamma, A) := \{a \in A \mid a^\sigma = a \text{for all } \sigma \in \Gamma\}.$$

A 1-*cocycle of* Γ *on* A is a map

$$\alpha : \Gamma \to A, \quad \sigma \mapsto \alpha_\sigma,$$

such that

$$\alpha_{\sigma\tau} = (\alpha_\sigma)^\tau \alpha_\tau \quad \text{for all } \sigma, \tau \in \Gamma.$$

Two cocycles α, β on A are called *cohomologous* (with respect to a) if there exists $a \in A$, such that $\beta_\sigma = a^{-\sigma} \cdot \alpha_\sigma \cdot a$ for all $\sigma \in \Gamma$. Note that being cohomologous is an equivalence relation.

We denote by $Z^1(\Gamma, A)$ the set of all 1-cocycles of Γ on A. We denote by $[\alpha]$ the equivalence class of α and by $H^1(\Gamma, A)$ the set of equivalence classes of 1-cocycles.

$Z^1(\Gamma, A)$ and $H^1(\Gamma, A)$ are *pointed sets*.

The constant map $t : \sigma \mapsto 1$ is the distinguished element of $Z^1(\Gamma, A)$, called the *trivial* 1-*cocycle*. Its cohomology class is the distinguished element of $H^1(\Gamma, A)$.

A twisted form A_β of A by the cocycle $\beta \in Z^1(\Gamma, A)$ is the same group A but with a different action of Γ on it, given by

$$a * \sigma := a^{\sigma\alpha_\sigma} \quad \text{for } \sigma \in \Gamma \text{ and } a \in A.$$

68.10.1 Creation of Gamma-groups

This section describes intrinsics dealing with cocycles and the first cohomology.

`GammaGroup(Gamma, A, action)`

Given a group A and a group Γ acting on it by the map *action*, return the object of type `GGrp`, which is the Group A together with this particular action of Γ. The map *action* must be a homomorphism from Γ to the automorphism group of A.

If B is a normal subgroup of A and normalised by the action of Γ on A (thus a Γ-group itself), then the action of Γ on A induces in the natural way to A/B. It is possible to create such a group:

`InducedGammaGroup(A, B)`

Given a Γ-group A and a normal subgroup B normalised by the action of Γ, return the induced Γ-group A/B.

Example H68E12

Let Γ act on A by conjugation:

```
> A := SymmetricGroup(4);
> Gamma := sub<A|(1,2,3), (1,2)>;
> action := hom< Gamma -> Aut(A) |
>             g :-> iso< A -> A | a :-> a^g, a :-> a^(g^-1) > >;
> A := GammaGroup( Gamma, A, action );
> A;
Gamma-group:  Symmetric group acting on a set of cardinality 4
Order = 24 = 2^3 * 3
(1, 2, 3, 4)
(1, 2)
Gamma-action: Mapping from: GrpPerm: $, Degree 4 to
Set of all automorphisms of GrpPerm: $, Degree 4, Order 2^3 * 3
given by a rule [no inverse]
Gamma:        Permutation group acting on a set of cardinality 4
(1, 2, 3)
(1, 2)
>
```

and B be a normal subgroup of A:

```
> B := AlternatingGroup(4);
```

```
> AmodB := InducedGammaGroup( A, B );
> AmodB;
Gamma-group:  Symmetric group acting on a set of cardinality 2
Order = 2
(1, 2)
(1, 2)
Gamma-action: Mapping from: GrpPerm: $, Degree 4, Order 2 * 3 to
Set of all automorphisms of GrpPerm: $, Degree 2, Order 2
given by a rule [no inverse]
Gamma:        Permutation group acting on a set of cardinality 4
Order = 6 = 2 * 3
(1, 2, 3)
(1, 2)
Induced from another Gamma-group
```

`IsNormalised(B, action)`

Returns `true` if the group B is normalised by the action *action*, where *action* is as above.

`IsInduced(AmodB)`

Returns `true` iff the Γ-group *AmodB* was created as an induced Γ-group. If it is, then the Γ-groups A, B, the projection and representative maps are returned as well.

68.10.2 Accessing Information

`Group(A)`

Returns the group A as a `Grp` object to be used in MAGMA.

`GammaAction(A)`

Returns the action of Γ on A as a map.

`ActingGroup(A)`

Returns the group Γ acting on A.

68.10.3 One Cocycles

`OneCocycle(A, imgs)`

`OneCocycle(A, alpha)`

`Check` BoolElt *Default :* `true`

If the map $\alpha : \Gamma \to A$ or the sequence *imgs* of images of the generators $\Gamma.1, ..., \Gamma.n$ defines a 1-cocycle, return the 1-cocycle. By default, the map is checked to define a 1-cocycle. If it doesn't, `OneCocycle` will abort with an error. This check can be disabled by setting the optional argument `Check` to `false`.

`TrivialOneCocycle(A)`

Return the trivial 1-cocycle.

`IsOneCocycle(A, imgs)`

`IsOneCocycle(A, alpha)`

Return true if the map $\alpha : \Gamma \to A$ or the sequence *imgs* of images of the generators $\Gamma.1, ..., \Gamma.n$ defines a 1-cocycle and `false` otherwise. If `true`, return the cocycle as the second argument.

Note that `IsOneCocycle` does not abort with an error in contrast to `OneCocycle` if the map does not define a cocycle.

`AreCohomologous(alpha, beta)`

Return `true` if and only if the 1-cocycles α and β are cohomologous. If they are, return the intertwining element as the second return value.

`CohomologyClass(alpha)`

Return the cohomology class of the 1-cocycle α.

`InducedOneCocycle(AmodB, alpha)`

`InducedOneCocycle(A, B, alpha)`

Given a 1-cocycle on A, return the induced 1-cocycle on $AmodB$. The second version will generate the induced Γ-group A/B first.

`ExtendedOneCocycle(alpha)`

`OnlyOne` BoolElt *Default :* `false`

Given a 1-cocycle on an induced Γ-group A/B, return the set of all non-cohomologous 1-cocycles on A, which induce to α. If the optional argument `OnlyOne` is `true`, the set will contain at most one 1-cocycle. If α is not extendible, the returned set is empty.

`ExtendedCohomologyClass(alpha)`

Given a 1-cocycle on an induced Γ-group A/B, return the the set of all non-cohomologous 1-cocycles on A, which induce to a cocycle in the cohomology class of α. If no such cocycles on A exist, the returned set is empty.

`GammaGroup(alpha)`

Return the Γ-group on which α is defined.

`CocycleMap(alpha)`

Return the `Map` object corresponding to α.

68.10.4 Group Cohomology

`Cohomology(A, n)`

Given a finite group A and an integer n (currently restricted to being 1) return the n-th cohomology group $H^n(\Gamma, A)$. Since the group A is not assumed to be abelian, only $n = 0, 1$ can be used. Currently, only $n = 1$ implemented. (The zero cohomology of A is the subgroup of A centralised by Γ and can be constructed using group theoretical methods available in Magma.)

`OneCohomology(A)`

Return the first cohomology $H^1(\Gamma, A)$. as a set of representatives of all cohomology classes. If the group A is abelian, existing code by Derek Holt is used (see Chapter 68). Otherwise use [Hal05].

`TwistedGroup(A, alpha)`

Given the Γ-group A and a 1-cocycle α on it, return the twisted group A_α.

Example H68E13

First, we create the group $A = D_8$. The returned group is the usual permutation group on the octagon. Γ is the Normaliser of A in S_8 and is acting by conjugation.

```
> A := DihedralGroup(8);
> Gamma := sub< Sym(8) | (1, 2, 3, 4, 5, 6, 7, 8),
>    (1, 8)(2, 7)(3, 6)(4, 5), (2, 4)(3, 7)(6, 8) >;
> A^Gamma eq A;
true
> Gamma;
Permutation group Gamma acting on a set of cardinality 8
Order = 32 = 2^5
    (1, 2, 3, 4, 5, 6, 7, 8)
    (1, 8)(2, 7)(3, 6)(4, 5)
    (2, 4)(3, 7)(6, 8)
> action := hom< Gamma -> Aut(A) |
>             g :-> iso< A -> A | a :-> a^g, a :-> a^(g^-1) > >;
```

```
> A := GammaGroup( Gamma, A, action );
```

Now let B be the center of A and create the induced Γ-group A/B:

```
> B := Center(Group(A));
> AmodB := InducedGammaGroup(A, B);
```

Create the trivial 1-cocycle on A/B and compute its cohomology class:

```
> triv := TrivialOneCocycle(AmodB);
> CohomologyClass( triv );
{@
    One-Cocycle
    defined by [
    Id($),
    Id($),
    Id($)
    ],
    One-Cocycle
    defined by [
    Id($),
    (1, 4)(2, 7)(3, 8)(5, 6),
    (1, 4)(2, 7)(3, 8)(5, 6)
    ],
    One-Cocycle
    defined by [
    (1, 4)(2, 7)(3, 8)(5, 6),
    Id($),
    (1, 4)(2, 7)(3, 8)(5, 6)
    ],
    One-Cocycle
    defined by [
    (1, 4)(2, 7)(3, 8)(5, 6),
    (1, 4)(2, 7)(3, 8)(5, 6),
    Id($)
    ]
@}
```

Pick one of the cocycles in this class and compute the intertwining element:

```
> alpha := Random($1);alpha;
One-Cocycle
defined by [
(1, 4)(2, 7)(3, 8)(5, 6),
(1, 4)(2, 7)(3, 8)(5, 6),
Id($)
]
> bo, a := AreCohomologous(alpha,triv);
> bo; a;
true
```

```
(1, 5)(2, 8)(3, 7)(4, 6)
```

Now create another cocycle on A/B and extend it to A:

```
> alpha := OneCocycle( AmodB,
>                [Group(AmodB)| (1, 7, 4, 2)(3, 5, 8, 6),
>                               (1, 2, 4, 7)(3, 6, 8, 5),
>                               1 ] );
> ExtendedOneCocycle(alpha);
{
    One-Cocycle
    defined by [
    (1, 4, 7, 2, 5, 8, 3, 6),
    (1, 2, 3, 4, 5, 6, 7, 8),
    Id($)
    ],
    One-Cocycle
    defined by [
    (1, 8, 7, 6, 5, 4, 3, 2),
    (1, 6, 3, 8, 5, 2, 7, 4),
    Id($)
    ]
}
```

Pick a cocycle β in this set and check if it really induces to α:

```
> beta := Rep($1);
> InducedOneCocycle(AmodB, beta) eq alpha;
true
```

Finally, create the twisted group A_β:

```
> A_beta := TwistedGroup(A, beta);
> A_beta;
Gamma-group:  Permutation group acting on a set of cardinality 8
Order = 16 = 2^4
(1, 2, 3, 4, 5, 6, 7, 8)
(1, 8)(2, 7)(3, 6)(4, 5)
Gamma-action: Mapping from: GrpPerm: $, Degree 8, Order 2^5 to
Set of all automorphisms of GrpPerm: $, Degree 8, Order 2^4
given by a rule [no inverse]
Gamma:        Permutation group acting on a set of cardinality 8
Order = 32 = 2^5
(1, 2, 3, 4, 5, 6, 7, 8)
(1, 8)(2, 7)(3, 6)(4, 5)
(2, 4)(3, 7)(6, 8)
>
```

68.11 Bibliography

[**CCH01**] J.J. Cannon, B. Cox, and D.F. Holt. Computing the subgroup lattice of a permutation group. *J. Symbolic Comp.*, 31:149–161, 2001.

[**Hal05**] Sergei Haller. *Computing Galois Cohomology and Forms of Linear Algebraic Groups*. Phd thesis, Technical University of Eindhoven, 2005.

[**Hol84**] D.F. Holt. The calculation of the Schur multiplier of a permutation group. In *Computational group theory (Durham, 1982)*, pages 307–319. Academic Press, London, 1984.

[**Hol85a**] D.F. Holt. A computer program for the calculation of a covering group of a finite group. *J. Pure Appl. Algebra*, 35(3):287–295, 1985.

[**Hol85b**] D.F. Holt. The mechanical computation of first and second cohomology groups. *J. Symbolic Comp.*, 1(4):351–361, 1985.