Solving Problems with Magma

Wieb Bosma
John Cannon
Catherine Playoust
Allan Steel

School of Mathematics and Statistics
University of Sydney
NSW 2006, Australia

Typeset by computer using Donald Knuth's TEX, and the document preparation system LATEX developed by Leslie Lamport.

Introduction

This book is neither an introductory manual nor a reference manual for MAGMA. Those needs are met by the books *An Introduction to Magma* and *Handbook of Magma Functions*. Even the most keen inductive learners will not learn all there is to know about MAGMA from the present work.

What *Solving Problems with Magma* does offer is a large collection of real-world algebraic problems, solved using the MAGMA language and intrinsics. It is hoped that by studying these examples, especially those in your specialty, you will gain a practical understanding of how to express mathematical problems in MAGMA terms. Most of the examples have arisen from genuine research questions, some of which the other computer algebra systems on the market cannot handle well, and a few which stretch MAGMA to its limit too. If you are trying the examples on your own MAGMA, be warned that some of the examples require significant CPU time and/or storage space to complete their execution.

We thank all those who have contributed examples. Some are older problems, solved with MAGMA's predecessor Cayley. Others are new ones that exploit MAGMA's ability to straddle all aspects of algebra. We welcome comments on this book and submissions of new approaches that you think might advantage other mathematicians.

Contents

Chapter 1

Language

1.1 Puzzle-solving

1.1.1 Dog Daze

The problem 'DOG DAZE' was taken from a May 1994 edition of the magazine published with the *Weekend Australian* newspaper. It is not difficult to solve by hand, but the following code, by Greg Gamble, indicates a way of solving it using Magma.

The Problem:

Devious Dan, the dogcatcher, found the five local dog owners most helpful, yet he still couldn't work out who Ms Green was, her dog's name, or even the kind of dog she had. Ms Brown's dog, Loopsie or Mooksie, was not the terrier. Woopsie, the poodle, did not belong to Mary. Marion's setter was not Loopsie. Margie's basset was called Smooksie or Mooksie. Myrtle, who was not Ms Grey, owned Poopsie or Smooksie. Martha Black didn't own the great dane. Ms Grey didn't own Poopsie, but Mooksie belonged to Ms White, who was not Marion.

The Solution:

The given information is translated into Magma code.

```
// Cartesian product of all combinations
> firstnames := {"Margie", "Marion", "Myrtle", "Martha", "Mary"};
> surnames := {"Brown", "Green", "Grey", "Black", "White"};
> dogbreeds := {"terrier", "poodle", "great dane", "basset", "setter"};
> dognames := {"Loopsie", "Mooksie", "Smooksie", "Woopsie", "Poopsie"};
> C := car<firstnames, surnames, dogbreeds, dognames>;

> // Some useful functions are defined:
> iff := 'eq'; // the '...' treats this operator as a function
> implies := func< A, B | B or not A >;

> S := { a : a in C |
>     implies(sn eq "Brown", (dn in {"Loopsie","Mooksie"})) and
>     implies(sn eq "Brown", (db ne "terrier")) and
>     iff(dn eq "Woopsie", db eq "poodle") and
>     implies(dn eq "Woopsie", fn ne "Mary") and
>     iff(fn eq "Marion", db eq "setter") and
```

```
>     implies(fn eq "Marion", dn ne "Loopsie") and
>     iff(fn eq "Margie", db eq "basset") and
>     implies(fn eq "Margie", (dn in {"Smooksie","Mooksie"})) and
>     implies(fn eq "Myrtle", sn ne "Grey") and
>     implies(fn eq "Myrtle", (dn in {"Poopsie","Smooksie"})) and
>     iff(fn eq "Martha", sn eq "Black") and
>     implies(fn eq "Martha", db ne "great dane") and
>     implies(sn eq "Grey", dn ne "Poopsie") and
>     iff(dn eq "Mooksie", sn eq "White") and
>     implies(sn eq "White", fn ne "Marion")
>         where fn, sn, db, dn is Explode(a) };
```

At this stage, the function needed to solve the problem can be defined. It returns the set of all possible solutions.

A solution of the problem is a subset of the set S consisting of five 4-tuples, such that each firstname etc. occurs precisely once. Observe that if `<fn,sn,db,dn>` is a member of a solution then there exist four 4-tuples among the set of 4-tuples in S not having firstname equal to `fn` etc. such that firstname etc. occurs precisely once. In this way the problem is reduced (recursively) to problems involving successively fewer 4-tuples.

```
> findsoln := function(S, n)
>    if n eq 1 then
>       return {S};
>    end if;
>    soln := {};
>    T := S;
>    while #T ge n do
>       ExtractRep(~T, ~x);
>       sol:=$$({a : a in T | forall{i : i in [1..4] | a[i] ne x[i] }}, n-1);
>       soln join:= { Include(U, x) : U in sol | #U eq n-1 };
>    end while;
>    return soln;
> end function;
```

With this function, the problem can be solved:

```
> findsoln(S, 5);
{
    { <Marion, Grey, setter, Smooksie>, <Myrtle, Green, terrier, Poopsie>,
    <Margie, White, basset, Mooksie>, <Mary, Brown, great dane, Loopsie>,
    <Martha, Black, poodle, Woopsie> }
}
```

The solution can be read from the output. Note that it is unique.

1.1.2 Letters standing for digits

The problem 'How many are whole?', by Susan Denham, was taken from *New Scientist* No. 1998 (7 Oct 1995), in the Enigma problem section on p. 63. It was brought to the attention of the

newsgroup `sci.math.symbolic` by Dr. Johan Wideberg. This solution is by Catherine Playoust, with Bruce Cox.

The Problem:

In what follows, digits have been consistently replaced by letters, with different letters being used for different digits:

In the list ONE TWO THREE FOUR just the first and one other are perfect squares.
In the list ONE + 1 TWO + 1 THREE + 1 FOUR + 1 just one is a perfect square.
In the list ONE + 2 TWO + 2 THREE + 2 FOUR + 2 just one is a perfect square.

If you want to win the prize send in your FORTUNE.

The Published Answer:

The answer published a few weeks later was FORTUNE = 3701284.

The Magma Solution:

Assuming that there are no leading zeros in ONE, TWO, THREE, FOUR, we know that none of O, T, F is zero. Since ONE is a perfect square and is greater than 100, it follows that ONE+1 and ONE+2 cannot be squares, so they can be put aside from the second and third conditions. Since TWO is greater than 100, at most one of TWO, TWO+1, TWO+2 can be a square, and similarly for THREE and FOUR. There are three conditions involving TWO, THREE, FOUR, so it follows that $x, y+1, z+2$ must all be perfect squares, for some permutation (x, y, z) of (TWO, THREE, FOUR). Therefore the conditions can be rephrased as shown below:

- ONE is a perfect square;
- exactly one of TWO, THREE, FOUR is a perfect square;
- exactly one of TWO+1, THREE+1, FOUR+1 is a perfect square;
- exactly one of TWO+2, THREE+2, FOUR+2 is a perfect square.

We construct all the possibilities for ONE explicitly, by finding those squares in the range 100 to 999 whose digits are distinct. The numbers with squares in this range are between `Ceiling(Sqrt(100))` = 10 and `Floor(Sqrt(999))` = 31. The following constructor builds the set of possibilities. Note that the intrinsic `Intseq(n, b)` converts the integer n to a base-b representation of it as a sequence starting with the units digit.

```
> ONEoptions := { Reverse(sq) :
>     sq in {Intseq(i^2, 10) : i in [10..31]} | #Set(sq) eq 3};
> #ONEoptions;
13
```

`ONEoptions` has size 13, so there are 13 possibilities for ONE. Let a particular choice from `ONEoptions` be `ONEchoice`.

At this stage, the solution splits into two: one slower algorithm, that is relatively easy to write; and one much faster algorithm, that takes some care in crafting.

Slower Algorithm: The digits `TFoptions` for {T, F} will be chosen as a subset of size 2 from the set of non-zero digits with `ONEchoice` removed (this set has size 6). The digits `WHRUoptions`

for {W, H, R, U} will be chosen as a subset of size 4 from the set of all digits with the digits for `ONEchoice` and `TFchoice` removed (this set has size 5). Finally, the ordered choice `TFchoice` for T, F will be made by permuting `TFoptions`, and similarly for `WHRUchoice`. Note that the total number of choices of digits considered under this scheme is

```
> 13 * Binomial(6, 2) * Binomial(5, 4) * Factorial(2) * Factorial(4);
46800
```

Now we can construct the solutions, in a single assignment. Note that in the second and third lines, the effect is to construct the permutations of `WHRUoptions` and `TFoptions`. The function `Subsets(T, k)` returns the sequence of all subsets of set T of size k.

```
> time solutions := {<O, N, E, T, F, W, H, R, U> :
>     WHRUchoice in Setseq(WHRUoptions) ^ SymmetricGroup(WHRUoptions),
>     TFchoice in Setseq(TFoptions) ^ SymmetricGroup(TFoptions),
>     WHRUoptions in Subsets({0..9} diff Seqset(ONEchoice) diff TFoptions, 4),
>     TFoptions in Subsets({1..9} diff Seqset(ONEchoice), 2),
>     ONEchoice in ONEoptions |
>     (IsSquare(TWO) or IsSquare(THREE) or IsSquare(FOUR))
>     and (IsSquare(TWO+1) or IsSquare(THREE+1) or IsSquare(FOUR+1))
>     and (IsSquare(TWO+2) or IsSquare(THREE+2) or IsSquare(FOUR+2))
>         where TWO is 100*T+10*W+O
>         where THREE is 10000*T+1000*H+100*R+11*E
>         where FOUR is 1000*F+100*O+10*U+R
>         where O, N, E is Explode(ONEchoice)
>         where T, F is Explode(TFchoice)
>         where W, H, R, U is Explode(WHRUchoice) };
Time: 10.690
> #solutions;
1
```

It emerges that the solution is unique. We check that it satisfies the conditions:

```
> soln := Rep(solutions); print soln;
<7, 8, 4, 1, 3, 6, 9, 0, 2>
> O, N, E, T, F, W, H, R, U := Explode(soln);
> ONE := 100*O+10*N+E;
> TWO := 100*T+10*W+O;
> THREE := 10000*T+1000*H+100*R+11*E;
> FOUR := 1000*F+100*O+10*U+R;
> ONE, TWO, THREE, FOUR;
784 167 19044 3720
> IsSquare(ONE) and IsSquare(TWO+2)
>     and IsSquare(THREE) and IsSquare(FOUR+1);
true
```

Finally, we calculate FORTUNE, as the question asked. The result matches the published answer:

```
> 1000000*F+100000*O+10000*R+1000*T+100*U+10*N+E;
3701284
```

Faster Algorithm: The reason that the above algorithm is slow is that it constructs all possible choices of digits such that ONE works, and then tests everything. It would be improved if the choice for TWO were tested before digits were allocated to THREE and FOUR, and similarly if the choice for THREE were be tested before digits were allocated to FOUR. (Testing FOUR as well makes little practical difference, since there are only a few options at that stage.) This method is followed in the version below:

```
> // construct ONEoptions as before
> ONEoptions := { Reverse(sq) :
>     sq in {Intseq(i^2, 10) : i in [10..31]} | #Set(sq) eq 3};
> time solutions := {<O, N, E, T, F, W, H, R, U> :
>     U in {0..9} diff Seqset(ONEchoice) diff {T, W, F, H, R},
>     R in {R : R in {0..9} diff Seqset(ONEchoice) diff {T, W, F, H} |
>         IsSquare(THREE) or IsSquare(THREE+1) or IsSquare(THREE+2)
>         where THREE is 10000*T+1000*H+100*R+11*ONEchoice[3] },
>     H in {0..9} diff Seqset(ONEchoice) diff {T, W, F},
>     F in {1..9} diff Seqset(ONEchoice) diff {T, W},
>     W in {W : W in Exclude({0..9}, T) diff Seqset(ONEchoice) |
>         IsSquare(TWO) or IsSquare(TWO+1) or IsSquare(TWO+2)
>         where TWO is 100*T+10*W+ONEchoice[1] },
>     T in {1..9} diff Seqset(ONEchoice),
>     ONEchoice in ONEoptions |
>     (IsSquare(TWO) or IsSquare(THREE) or IsSquare(FOUR))
>     and (IsSquare(TWO+1) or IsSquare(THREE+1) or IsSquare(FOUR+1))
>     and (IsSquare(TWO+2) or IsSquare(THREE+2) or IsSquare(FOUR+2))
>         where TWO is 100*T+10*W+O
>         where THREE is 10000*T+1000*H+100*R+11*E
>         where FOUR is 1000*F+100*O+10*U+R
>         where O, N, E is Explode(ONEchoice) };
Time: 0.139
> solutions;
{ <7, 8, 4, 1, 3, 6, 9, 0, 2> }
```

The answer is the same as before, but the execution time is much less.

1.2 Sets, sequences and functions

1.2.1 Farey sequence

The Farey series F_n of degree n consists of all rational numbers with denominator less than or equal to n, in order of magnitude. Since we will need the `Numerator` and `Denominator` functions often, we first define abbreviations for them.

```
> D := Denominator;
> N := Numerator;
```

We present three ways to obtain the Farey series F_n of degree n. The CPU time taken to calculate F_{100} will be measured for each version, in order to compare the algorithms.

The first method calculates the entries in order. It uses the fact that for any three consecutive Farey fractions $\frac{p}{q}$, $\frac{p'}{q'}$, $\frac{p''}{q''}$ of degree n:

$$p'' = \lfloor \frac{q+n}{q'} \rfloor p' - p, \quad q'' = \lfloor \frac{q+n}{q'} \rfloor q' - q.$$

```
> farey := function(n)
>      f := [ RationalField() | 0, 1/n ];
>      repeat
>          p := ( D(f[#f-1]) + n) div D(f[#f]) * N(f[#f])  - N(f[#f-1]);
>          q := ( D(f[#f-1]) + n) div D(f[#f]) * D(f[#f])  - D(f[#f-1]);
>          Append(~f, p/q);
>      until p/q ge 1;
>      return f;
> end function;
> farey(6);
[ 0, 1/6, 1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 1 ]
> time f100 := farey(100);
Time: 0.389
```

The second method calculates the Farey series recursively. It uses the property that F_n may be obtained from F_{n-1} by inserting a new fraction (namely $\frac{p+p'}{q+q'}$) between any two consecutive rationals $\frac{p}{q}$ and $\frac{p'}{q'}$ in F_{n-1} for which $q + q'$ equals n.

```
> function farey(n)
>      if n eq 1 then
>          return [RationalField() | 0, 1 ];
>      end if;
>      f := farey(n-1);
>      i := 0;
>      while i lt #f-1 do
>          i +:= 1;
>          if D(f[i]) + D(f[i+1]) eq n then
>              Insert(~f, i+1, (N(f[i]) + N(f[i+1]))/(D(f[i]) + D(f[i+1])));
>          end if;
>      end while;
>      return f;
> end function;
> time f100 := farey(100);
Time: 3.909
```

The third method uses `Setseq` to convert a set into a sequence and `Sort` to put the terms of the sequence in ascending order:

```
> farey := func< n | Sort(Setseq({ a/b : a in {0..n}, b in {1..n} | a le b }))>;
> time f100 := farey(100);
Time: 0.130
```

The third method was fastest, followed by the first method. The recursive method was much slower, but note that it effectively calculates all the Farey sequences up to the given n.

1.2.2 The knapsack problem

The knapsack problem is concerned with the task of selecting items to be packed into a knapsack. The items are chosen from a large number of objects with different volumes. One can choose as many items of whatever volume to go into the knapsack as one likes, provided that the total volume of the knapsack is filled. This problem is in the class of NP-complete problems.

Here we present a Magma function that produces a solution to the knapsack problem. It operates on the pseudo-non-deterministic principle of choosing some object to go into the knapsack and trying to pack the rest of the knapsack, recursively. The volume of the knapsack is t, and the list of volumes of the objects is contained in the sequence Q. The aim is to find some entries of Q whose sum is t.

```
> knapsack := function(Q, t)
>     R := [IntegerRing() | x : x in Q | x le t];
>     if &+R lt t then
>         return [ ];
>     elif t in R then
>         return [t];
>     else
>         for x in R do
>             RR := Exclude(R, x);
>             s := $$(RR, t - x);
>             if not IsEmpty(s) then
>                 return Append(s, x);
>             end if;
>         end for;
>         return [ ];
>     end if;
> end function;
```

We use this function to find some distinct primes less than 100 whose sum is 151:

```
> primes := [ p : p in [1..100] | IsPrime(p) ];
> k := knapsack(primes, 151);
> k, &+k;
[ 43, 31, 19, 17, 13, 11, 7, 5, 3, 2 ]
151
```

1.2.3 Simulation of a cellular automaton

This problem has been adapted from a Mathematica example discussed by Richard Gaylord.[1]The Magma program was developed by Graham Matthews. Thanks also to Richard J. Fateman for his assistance.

Introduction: Turbulent cascading has been suggested as the underlying cause of a wide variety of phenomena, including geological upheavals (such as volcanic eruptions and earthquakes), species

[1]Richard J. Gaylord and Paul R. Wellin, *Computer Simulations with Mathematica: Explorations in Complex Physical and Biological Systems* (Santa Clara, CA: Springer-Verlag TELOS, 1995), 147–149.

extinction during evolution, and fluid turbulence. A simple one-dimensional probabilistic cellular automaton (CA), known as the forest fire model, displays this behaviour.

The Forest Fire CA System: The forest fire CA employs a one-dimensional lattice of length n, with periodic boundary conditions. Lattice sites may have a value of 0, 1 or 2, where 0 represents an empty site (or 'hole'), 1 represents a healthy tree (or 'tree'), and 2 represents a burning tree. A forest is a set of contiguous sites (i.e., a connected segment) with value 1 or 2. A forest preserve consists of forests separated by gaps (or 'deserts') consisting of clusters of connected holes. The system evolves in a specified number of time steps, in each of which entire forests of trees can catch fire and burn down and trees can sprout on individual empty sites.

(Note: having all of the trees in a forest burn down in a single time step while trees grows independently of one another provides a separation between the time scales for the processes of deforestation and reforestation).

The Algorithm:

(1) A forest preserve of length n, consisting of empty sites and tree sites, is created.

All of the sites in the forest preserve are updated in each time step, according to the following sequence of steps:

(2a) Trees catch fire with probability f and empty sites sprout trees with probability p.

(2b) All tree sites adjacent to an ignited tree site (i.e., trees in the same forest) ignite.

(2c) Ignited trees burn down, becoming holes.

The sequence of steps 2a-c can be combined and applied to any forest preserve configuration.

(3) Step 2 is repeated m times.

The Magma Program:

```
> Smokey := function(n, p, f, m)
>     InitialPreserve := [ Random(0,1) : i in [1..n] ];
>
>     treeGrowIgnite :=
>       func< F | [ x eq 0 select Floor(1+p-Random(0,1))
>                         else Floor(2+f-Random(0,1)) : x in F ] >;
>
>     forestIgnite := function(F)
>         ignite := func< n1, a, n2 |
>           a eq 1 and (n1 eq 2 or n2 eq 2) select 2 else a >;
>         next := func< F |
>           [ ignite(F[#F], F[1], F[2]) ] cat
>           [ ignite(F[i-1], F[i], F[i+1]) : i in [2..#F-1] ] cat
>           [ ignite(F[#F-1], F[#F], F[1]) ] >;
>         // compute the result of forestIgnite as a fixed point
>         return func< F1,F2 |
>           F1 eq F2 select F1 else $$(F2, next(F2)) >(F, next(F));
>     end function;
>
>     destroy := func< F | [ x eq 2 select 0 else x : x in F ] >;
```

```
>
>     return [ i eq 1 select InitialPreserve else
>       destroy(forestIgnite(treeGrowIgnite(Self(i-1)))) : i in [1..m]];
> end function;
```

We can call the function for a forest of 25 trees, with a probability of catching fire of 0.2 and a probability of empty sites sprouting trees of 0.4, over 200 iterations:

```
> F := Smokey(25, 0.2, 0.4, 200);
```

Now the sequence F contains the history of the preserve. By printing some values of F we can see how the forest changes over the iterations.

```
> F[1], F[5], F[10], F[50], F[100], F[200];
[ 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1 ]
[ 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0 ]
[ 0, 0, 0, 0, 1, 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, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1 ]
[ 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1 ]
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0 ]
```

Chapter 2

The Integers

2.1 Introduction

Magma contains a purpose-written fast integer arithmetic package. Both Karatsuba and FFT algorithms are used for integer multiplication while the Weber accelerated GCD algorithm is used for GCD calculation. Assembler macros are used for critical operations and 64-bit hardware instructions are used on DEC-Alpha machines.

2.2 Arithmetic and Arithmetic Functions

- Karatsuba and FFT algorithms for integer multiplication
- Karatsuba-based algorithm for integer division
- Weber's algorithm for GCD calculation
- Arithmetic functions: Jacobi symbol, Euler ϕ function, etc.
- Alternative representation of integers in factored form
- Modular arithmetic: Exponentiation, square root, order, primitive element
- Chinese remainder theorem

2.2.1 Example: Amicable numbers

A pair of positive integers (m, n) is called *amicable* if n is equal to the sum of the proper divisors of m (that is, the divisors excluding m itself), and vice versa. The following code finds such pairs. Note that it also finds perfect numbers, that is, amicable pairs of the form (m, m).

```
> d := func< m | DivisorSigma(1, m) - m >;
> for m := 2 to 10000 do
>     n := d(m);
>     if m ge n and d(n) eq m then
>         print m, n;
>     end if;
> end for;
6 6
```

```
28 28
220 284
284 220
496 496
1184 1210
1210 1184
2620 2924
2924 2620
5020 5564
5564 5020
6232 6368
6368 6232
8128 8128
```

2.3 Factorization and Primality Proving

- Elementary factorization techniques: Trial division, SQUFOF, Pollard ρ, Pollard $p-1$
- Elliptic curve method for integer factorization (A. Lenstra)
- Multiple prime multiple polynomial quadratic sieve algorithm for integer factorization (A. Lenstra)
- Database of factorizations of integers of the form $p^n \pm 1$
- Primality testing (Miller-Rabin)
- Primality proofs (Morain's Elliptic Curve Primality Prover), primality certificates
- Generation of primes

2.3.1 Example: Cunningham Factorization

Magma contains a database of factorizations of 199,044 integers of the form $a^n \pm 1$. This database includes the factorizations of integers of the form $a^n \pm 1$, $a \leq 12$, produced by Sam Wagstaff and collaborators. In addition, it includes factorizations of integers of the form $p^n \pm 1$, for primes $13 \leq p \leq 1000$ as tabulated by Richard Brent and collaborators. Given an integer of the form $a^n \pm 1$ that lies in the scope of the tables, Magma empolys an algorithm due to Richard Brent that derives its factorization using a combination of table lookup, Aurifeuillian factorization and algebraic factorization. We demonstrate the process by factoring $3^{429}+1$, an integer having 205 decimal digits.

```
> SetVerbose("Cunningham", true);
> n := 3^429 + 1;
> Factorization(n);
Cunningham Factorization: a = 3, e = 429, c = 1
Trial Division
Found prime <2, 2>
Found prime <7, 1>
Found prime <67, 1>
Found prime <79, 1>
Found prime <157, 1>
Found prime <661, 1>
```

```
Found prime <859, 1>
Found prime <2887, 1>
Database lookup
Found prime <53197, 1>
Found prime <20593, 1>
Found prime <1680823, 1>
Found prime <33531499, 1>
Found prime <207344547828773831191, 1>
Aurifeuillian factors
Found prime <176419, 1>
Found prime <25411, 1>
Found prime <10141, 1>
Found prime <1015425070461617058397746329107522771307833, 1>
Found prime <119655082715674302024296547198 73, 1>
Algebraic factors
Found prime <398581, 1>
Primality testing 450401094967106229593918726033137064469216450527907 73
Found prime <450401094967106229593918726033137064469216450527907 73, 1>
[ <2, 2>, <7, 1>, <67, 1>, <79, 1>, <157, 1>, <661, 1>, <859, 1>, <2887, 1>,
<10141, 1>, <20593, 1>, <25411, 1>, <53197, 1>, <176419, 1>, <398581, 1>,
<1680823, 1>, <33531499, 1>, <207344547828773831191, 1>,
<119655082715674302024296547198 73, 1>,
<1015425070461617058397746329107522771307833, 1>,
<450401094967106229593918726033137064469216450527907 73, 1> ]
Time: 2.229
```

2.3.2 Example: Sums of Squares

For each integer n in the range $10^{20} \leq n \leq 10^{20} + 100$, we determine whether n can be written as the sum of two perfect squares. The function `NormEquation` is used for the computation.

```
> b := 10^20;
> for n in [b .. b + 100] do
>     s, x, y := NormEquation(1, n);
>     if s then
>         printf "%o = %o^2 + %o^2\n", n, x, y;
>     end if;
> end for;
100000000000000000000 = 10000000000^2 + 0^2
100000000000000000001 = 10000000000^2 + 1^2
100000000000000000004 = 10000000000^2 + 2^2
100000000000000000009 = 9551166965^2 + 2962298028^2
100000000000000000016 = 10000000000^2 + 4^2
100000000000000000025 = 8429062912^2 + 5380603909^2
100000000000000000036 = 9694446870^2 + 2453100056^2
100000000000000000045 = 8337268941^2 + 5521770242^2
100000000000000000049 = 9969472985^2 + 780774232^2
100000000000000000053 = 9421472622^2 + 3351992487^2
100000000000000000058 = 8964092203^2 + 4432273793^2
100000000000000000064 = 10000000000^2 + 8^2
100000000000000000072 = 9727676814^2 + 2317823074^2
100000000000000000080 = 8585762712^2 + 5126858556^2
```

```
10000000000000000081 = 8823529416^2 + 4705882345^2
10000000000000000084 = 9918248430^2 + 1276067428^2
10000000000000000085 = 9995375401^2 + 304089778^2
10000000000000000093 = 9792534693^2 + 2026391938^2
10000000000000000097 = 8802221156^2 + 4745619319^2
10000000000000000100 = 8011983994^2 + 5983988008^2
```

Chapter 3

Univariate Polynomial rings

3.1 Introduction

A polynomial ring may be formed over any commutative ring, including a polynomial ring. Since computational methods for univariate polynomial rings are often much simpler and more efficient than those for multivariate rings (especially over a field), we discuss the two cases separately. In this section, the symbol K, appearing as a coefficient ring, will denote a field.

3.2 Univariate Polynomial Rings: Creation and Ring Operations

- Creation of a polynomial ring over any commutative ring
- Homomorphisms from polynomial rings into other rings

3.3 Univariate Polynomial Rings: Arithmetic with Polynomials

- Arithmetic with elements (Karatsuba algorithm for multiplication)
- Determination of whether an element is: a unit, a zero-divisor, nilpotent
- Differentiation and integration

3.4 Univariate Polynomial Rings: GCD and Resultant

- Resultant (sub-resultant algorithm, Euclidean algorithm), discriminant
- Greatest common divisor (modular and GCD-HEU algorithms for $\mathbf{Z}$)

3.4.1 Example: Resultant and GCD

The resultant of two polynomials tells whether they have a non-trivial GCD. We take two polynomials in $\mathbf{Z}[x]$ which are coprime and using the resultant we find the primes modulo which the polynomials have a common factor.

```
> P<x> := PolynomialRing(IntegerRing());
> f := x^50 - x + 1;
> g := x^41 - x^7 + 3;
> GCD(f, g);
1
> r := Resultant(f, g);
> r;
823751968381209779686428
```

For each prime divisor p of the resultant r we note the GCD of the polynomials modulo p.

```
> PrimeDivisors(r);
[ 2, 3, 71, 704807, 1584901, 865534777 ]
> for p in PrimeDivisors(r) do
for> PP<y> := PolynomialRing(GF(p));
for> printf "p = %o, GCD = %o\n", p, GCD(PP ! f, PP ! g);
for> end for;
p = 2, GCD = y^2 + y + 1
p = 3, GCD = y + 1
p = 71, GCD = y + 40
p = 704807, GCD = y + 476610
p = 1584901, GCD = y + 297038
p = 865534777, GCD = y + 337462522
```

For all other primes not dividing the resultant, the GCD is trivial. We check that this is the case for the primes up to 23.

```
> for p in [5, 7, 11, 13, 17, 19, 23] do
for> PP<y> := PolynomialRing(GF(p));
for> printf "p = %o, GCD = %o\n", p, GCD(PP ! f, PP ! g);
for> end for;
p = 2, GCD = y^2 + y + 1
p = 3, GCD = y + 1
p = 5, GCD = 1
p = 7, GCD = 1
p = 11, GCD = 1
p = 13, GCD = 1
p = 17, GCD = 1
p = 19, GCD = 1
p = 23, GCD = 1
```

3.4.2 Example: Modular GCD algorithm

We give a Magma language implementation of the modular algorithm for computing the GCD of two monic polynomials with integer coefficients. [A description of the algorithm may be found in J.H. Davenport, Y. Siret and E. Tournier, *Computer Algebra*, Academic Press, London, 1988.] (Magma has an intrinsic `GCD` to compute this value; the example is for illustrative purposes). The GCD function, `ModGCD`, calls the function `Chinese` to perform the Chinese Remainder Algorithm on the polynomials a modulo m and b modulo p. `Chinese`, in turn, calls the Magma intrinsic `Solution(u, v, m)`, where u, v and m are pairs of integers, to solve the congruences

$$u_1 x = v_1 \pmod{m_1}, \qquad u_2 x = v_2 \pmod{m_2},$$

where m_1 and m_2 are coprime.

```
> Chinese := function(a, m, b, p)
>     X := [ Solution([1, 1], [Coefficient(a, i), Coefficient(b, i)], [m, p]) :
>             i in [0..Max(Degree(a), Degree(b))] ];
>     return Parent(a) ! [ x gt (p*m) div 2 select x-p*m else x : x in X ];
> end function;
```

The polynomial a and the polynomial returned by `Chinese` are equal when both are coerced into the ring of polynomials over `ResidueClassRing(m)`, and similarly for b and p.

For each successive prime, the function `ModGCD` creates $\mathbf{F}_p[u]$ together with the natural homomorphisms $\phi : \mathbf{Z}[x] \to \mathbf{F}_p[u]$ and $\rho : \mathbf{F}_p[u] \to \mathbf{Z}[x]$. The GCD of the images of polynomials f and g in $\mathbf{F}_p[u]$ is found by calling the intrinsic function `GCD`.

```
> ModGCD := function(f, g)
>     R<x> := Parent(f);
>     p := 2;
>     d := R ! 1;
>     m := 1;
>     repeat
>         p := NextPrime(p);
>         S<u> := PolynomialRing(FiniteField(p));
>         phi  := hom< R -> S | x -> u >;
>         rho  := hom< S -> R | u -> x >;
>         e := GCD(phi(f), phi(g));
>         if Degree(e) lt Degree(d) then
>             d := Chinese(R!1, 1, rho(e), p);
>             m := p;
>         else
>             d := Chinese(d, m, rho(e), p);
>             m *:= p;
>         end if;
>     until (f mod d eq 0) and (g mod d eq 0);
>     return d;
> end function;
```

We apply our function to a pair of monic polynomials with integer coefficients:

```
> R<x> := PolynomialRing(IntegerRing());
```

```
> f := (x^17 - 5*x^9 + 10)^8 * (x^2 - 4*x + 4)^3 * (x^2 + 1);
> g := (x^2 + x + 1)^15 * (x - 2)^2 * (x - 7) * (x^17 - 5*x^9 + 10);
> gcd := ModGCD(f, g); print gcd;
x^19 - 4*x^18 + 4*x^17 - 5*x^11 + 20*x^10 - 20*x^9 + 10*x^2 - 40*x + 40
> gcd eq GCD(f, g); // compare with Magma intrinsic
true
```

We also try our function on a much larger example. The method is quite powerful even for very large examples.

```
> f := (x+1)^100;
> g := (x+2)^100;
> gcd := ModGCD(f*g, f*(g+1));
> time gcd := ModGCD(f*g, f*(g+1));
Time: 1.799
> gcd eq f;
true
> Degree(f);
100
> Max(Coefficients(f));
100891344545564193334812497256 51
```

3.5 Univariate Polynomial Rings: Factorization

- Factorization over GF(q): Small field Berlekamp, large field Berlekamp, Cantor-Zassenhaus algorithms
- Factorization over $\mathbb{Z}$ and $\mathbb{Q}$: Collins-Encarnacion algorithm
- Factorization over $\mathbb{Q}_p$: Ford-Zassenhaus algorithm
- Factorization over $\mathbb{Q}(\alpha)$: Trager algorithm

The key algorithms for univariate polynomials are GCD and factorization. Greatest common divisors for polynomials over $\mathbb{Z}$ are computed using either a modular algorithm or the GCD-HEU method, while for polynomials over a number field, a modular method is used. The factorization of polynomials over $\mathbb{Z}$ uses single factor bounds, parallel Hensel lifting and other recent ideas of Collins and Encarnacion. Magma factors the first challenge polynomial of Zimmermann (degree 156, 78 digit coefficients over $\mathbb{Z}$) in 6 seconds.

3.5.1 Example: Factorization over finite fields

We first factorize the polynomial $x^{100} + x + 1$ over the finite field of cardinality 2.

```
> K<w> := GF(2);
> P<x> := PolynomialRing(K);
> time Factorization(x^100 + x + 1);
[
    <x^14 + x^12 + x^10 + x^9 + x^5 + x^4 + 1, 1>,
```

```
    <x^17 + x^15 + x^13 + x^11 + x^6 + x^5 + x^4 + x^2 + 1, 1>,
    <x^69 + x^65 + x^64 + x^63 + x^62 + x^61 + x^59 + x^58 + x^53 + x^50 + x^48
        + x^45 + x^44 + x^43 + x^41 + x^39 + x^34 + x^33 + x^28 + x^25 + x^24 +
        x^23 + x^20 + x^19 + x^18 + x^15 + x^13 + x^10 + x^9 + x^8 + x^6 + x^5 +
        x^4 + x^3 + x^2 + x + 1, 1>
]
Time: 0.000
```

We now factorize the same polynomial over the finite field of cardinality 2^{14} and notice that the degree 14 factor above splits into linear factors.

```
> K<w> := GF(2, 14);
> P<x> := PolynomialRing(K);
> time Factorization(x^100 + x + 1);
[
    <x + w^3007, 1>,
    <x + w^6014, 1>,
    <x + w^7673, 1>,
    <x + w^8087, 1>,
    <x + w^9695, 1>,
    <x + w^12028, 1>,
    <x + w^12235, 1>,
    <x + w^13039, 1>,
    <x + w^14309, 1>,
    <x + w^14711, 1>,
    <x + w^15346, 1>,
    <x + w^15547, 1>,
    <x + w^15965, 1>,
    <x + w^16174, 1>,
    <x^17 + x^15 + x^13 + x^11 + x^6 + x^5 + x^4 + x^2 + 1, 1>,
    <x^69 + x^65 + x^64 + x^63 + x^62 + x^61 + x^59 + x^58 + x^53 + x^50 + x^48
        + x^45 + x^44 + x^43 + x^41 + x^39 + x^34 + x^33 + x^28 + x^25 + x^24 +
        x^23 + x^20 + x^19 + x^18 + x^15 + x^13 + x^10 + x^9 + x^8 + x^6 + x^5 +
        x^4 + x^3 + x^2 + x + 1, 1>
]
Time: 0.229
```

3.5.2 Example: Factorization over number fields

We factorization a polynomial of degree 40 over the cyclotomic field $K = \mathbb{Q}(\zeta_5)$ of order 5. Notice that the polynomial has rational coefficients but splits over K into polynomials whose coefficients are not rational.

```
> K<z> := CyclotomicField(5);
> P<x> := PolynomialRing(K);
> f :=
>  x^40 - 10*x^39 + 55*x^38 - 220*x^37 + 715*x^36 - 1992*x^35 + 4905*x^34 -
>     10890*x^33 + 22110*x^32 - 41470*x^31 + 72407*x^30 - 118390*x^29 +
>     182070*x^28 - 263780*x^27 + 357885*x^26 - 451170*x^25 + 535125*x^24 -
>     642750*x^23 + 888250*x^22 - 1383250*x^21 + 2015761*x^20 - 2289680*x^19
>     + 1551040*x^18 + 291240*x^17 - 2365605*x^16 + 3341910*x^15 -
```

```
>     2568525*x^14 + 740150*x^13 + 927450*x^12 - 1667600*x^11 + 1430378*x^10
>     - 684320*x^9 + 104685*x^8 + 59910*x^7 - 22470*x^6 + 2064*x^5 - 980*x^4
>     + 640*x^3 + 40*x^2 - 80*x + 16;
> time L := Factorization(f);
Time: 0.840
> L;
[
    <x^2 + 2*z*x + z^2 + 1, 1>,
    <x^2 + 2*z*x + z^2 + z, 1>,
    <x^2 + 2*z*x + 2*z^2, 1>,
    <x^2 + 2*z*x - z^3 - z - 1, 1>,
    <x^2 + 2*z*x + z^3 + z^2, 1>,
    <x^2 + 2*z^2*x - z^2 - z - 1, 1>,
    <x^2 + 2*z^2*x - 2*z^3 - 2*z^2 - 2*z - 2, 1>,
    <x^2 + 2*z^2*x - z^3 - z^2 - z, 1>,
    <x^2 + 2*z^2*x - z^3 - z^2 - 1, 1>,
    <x^2 + 2*z^2*x - z^3 - z - 1, 1>,
    <x^2 + (-2*z^3 - 2*z^2 - 2*z - 2)*x - z^2 - z - 1, 1>,
    <x^2 + (-2*z^3 - 2*z^2 - 2*z - 2)*x + z^3 + 1, 1>,
    <x^2 + (-2*z^3 - 2*z^2 - 2*z - 2)*x + z^3 + z, 1>,
    <x^2 + (-2*z^3 - 2*z^2 - 2*z - 2)*x + z^3 + z^2, 1>,
    <x^2 + (-2*z^3 - 2*z^2 - 2*z - 2)*x + 2*z^3, 1>,
    <x^2 + 2*z^3*x + z + 1, 1>,
    <x^2 + 2*z^3*x + 2*z, 1>,
    <x^2 + 2*z^3*x + z^2 + z, 1>,
    <x^2 + 2*z^3*x - z^3 - z^2 - 1, 1>,
    <x^2 + 2*z^3*x + z^3 + z, 1>
]
> // Check factorization is correct:
> &*[t[1]^t[2]: t in L] eq f;
true
```

Chapter 4

Finite Fields

4.1 Introduction

The finite field module uses different representations of finite field elements depending upon the size of the field. Thus, in the case of small to medium sized fields, the Zech logarithm representation is used. For a large degree extension K of a (small) prime field, K is represented as an extension of an intermediate field F whenever possible. The intermediate field F is chosen to be small enough so that the fast Zech logarithm representation may be used. Thus, Magma supports finite fields ranging from GF(2^n), where the degree n may be a thousand or more, to fields GF(p), where the characteristic p may be a thousand-bit integer.

4.2 Finite Fields

- Construction of fields GF(p), p large; GF(p^n), p small and n large
- Optimized representations of GF(p^n) in the case of small p and large n
- Construction of towers of extensions
- Arithmetic; relative trace and norm, order
- Testing elements for: Normality, primitivity
- Characteristic and minimum polynomials
- Square root (Tonelli-Shanks method for GF(p)), n-th root
- Construction and compatible embedding of subfields
- Construction of primitive and normal elements
- Enumeration of irreducible polynomials
- Polhig-Hellman method for discrete logarithms
- Field as an algebra over a subfield

4.2.1 Example: Lattice of Finite Fields

We create GF(5^{36}) and its full subfield-lattice and then experiment with the relationships. We start by creating $F36$ with generator $w36$ as GF(5^{36}) and note the defining polynomial of $F36$ (the minimal polynomial of $w36$).

```
> F36<w36> := FiniteField(5, 36);
> d<x> := DefiningPolynomial(F36);
> d;
x^36 + 4*x^33 + x^32 + 3*x^31 + 2*x^29 + 2*x^28 + x^27 + 2*x^26 + 4*x^25 + x^24
    + x^23 + 4*x^20 + x^18 + 4*x^15 + 4*x^13 + 3*x^12 + 2*x^11 + 3*x^10 + 2*x^9
    + 2*x^8 + 2*x^7 + x^6 + x^5 + 3*x^4 + 4*x^3 + 3*x + 2
```

Create all subfields of $F36$:

```
> F18<w18> := sub<F36 | 18>;
> F12<w12> := sub<F36 | 12>;
> F9<w9> := sub<F36 | 9>;
> F6<w6> := sub<F36 | 6>;
> F4<w4> := sub<F36 | 4>;
> F3<w3> := sub<F36 | 3>;
> F2<w2> := sub<F36 | 2>;
> F1 := sub<F36 | 1>;
```

Note that the defining polynomial of $F12$ is not the same as the Conway polynomial for GF(5^{12}).

```
> ConwayPolynomial(5, 12);
x^12 + x^7 + x^6 + 4*x^4 + 4*x^3 + 3*x^2 + 2*x + 2
> DefiningPolynomial(F12);
x^12 + 4*x^11 + x^10 + 3*x^9 + 3*x^8 + 3*x^7 + 4*x^5 + 3*x^3 + 4*x^2 + 3*x + 2
```

So set G with generator g to be the finite field whose defining polynomial is the Conway polynomial.

```
> G<g> := ext< F1 | ConwayPolynomial(5, 12) >;
> DefiningPolynomial(G);
x^12 + x^7 + x^6 + 4*x^4 + 4*x^3 + 3*x^2 + 2*x + 2
```

Now tell Magma to embed $F12$ in G.

```
> Embed(F12, G);
```

Note what g looks like in $F12$.

```
> F12 ! g;
4*w12^11 + w12^10 + 2*w12^7 + 2*w12^6 + 3*w12^5 + 3*w12^4 + 3*w12^3 + 3*w12^2 +
    4*w12 + 1
```

Check that the minimal polynomial of g over $F6$ is quadratic (note that $F6$ is automatically a subfield of G by transitivity).

```
> m<x6> := MinimalPolynomial(g, F6);
> m;
x6^2 + w6^7484*x6 + w6^3281
```

Since g is primitive (root of a Conway polynomial), setting h to the following power will obtain an element whose minimal polynomial has degree 6, i.e. h generates a field of degree 6.

```
> h := g^((5^12 - 1) div (5^6 - 1));
> h;
g^11 + 2*g^10 + g^9 + 2*g^8 + g^7 + 2*g^5 + 4*g^3 + 4*g^2 + 4*g + 3
```

Since all fields of the same degree are equal, h must be in $F6$. Set $h6$ to the element in $F6$ equal to h.

```
> h in F6;
true
> h6 := F6 ! h;
> h6;
w6^3281
```

Now lift h into $F36$ in two different ways and check that the "diagram" commutes. Lifting $h6$ via $F18$ gives the same answer as lifting h directly into $F36$.

```
> F18 ! h6;
2*w18^17 + 2*w18^16 + 4*w18^15 + w18^14 + 2*w18^13 + w18^11 + 3*w18^10 + 4*w18^8
    + w18^7 + 4*w18^6 + w18^5 + 2*w18^4 + 2*w18^3 + w18^2
> F36 ! (F18 ! h6);
4*w36^35 + 4*w36^34 + 4*w36^33 + 2*w36^32 + 3*w36^31 + 2*w36^30 + w36^29 +
    2*w36^27 + 2*w36^26 + w36^25 + 2*w36^24 + 2*w36^23 + 4*w36^22 + 3*w36^20 +
    2*w36^19 + w36^18 + 2*w36^17 + 3*w36^16 + 4*w36^14 + w36^12 + 4*w36^11 +
    4*w36^10 + w36^9 + w36^8 + 2*w36^7 + 3*w36^6 + 4*w36^5 + 2*w36^3 + 4*w36^2 +
    4*w36
> F36 ! h;
4*w36^35 + 4*w36^34 + 4*w36^33 + 2*w36^32 + 3*w36^31 + 2*w36^30 + w36^29 +
    2*w36^27 + 2*w36^26 + w36^25 + 2*w36^24 + 2*w36^23 + 4*w36^22 + 3*w36^20 +
    2*w36^19 + w36^18 + 2*w36^17 + 3*w36^16 + 4*w36^14 + w36^12 + 4*w36^11 +
    4*w36^10 + w36^9 + w36^8 + 2*w36^7 + 3*w36^6 + 4*w36^5 + 2*w36^3 + 4*w36^2 +
    4*w36
> F36 ! (F18 ! h6) eq F36 ! h;
true
```

Note that elements in two different fields ($w18$ in $F18$ and g in G) can be compared, combined etc. because there is an automatic cover for them.

```
> w18 + g;
w36^35 + 2*w36^34 + 2*w36^33 + w36^32 + 2*w36^31 + w36^30 + w36^29 + w36^27 +
```

```
    3*w36^25 + w36^24 + w36^23 + w36^22 + 4*w36^21 + 3*w36^20 + 2*w36^19 +
    4*w36^17 + 2*w36^16 + 3*w36^15 + w36^13 + 3*w36^11 + 3*w36^10 + 2*w36^9 +
    4*w36^8 + 4*w36^7 + 3*w36^6 + 4*w36^5 + 2*w36^4 + w36^3 + 4*w36^2 + 4*w36 + 3
```

Let z be $w18 + g$ (which will reside in $F36$). We can calculate the order of z and the factorization of the order of z easily.

```
> z := w18 + g;
> Order(z);
181898940354856475830078
> FactoredOrder(z);
[ <2, 1>, <3, 3>, <7, 1>, <13, 1>, <19, 1>, <31, 1>, <37, 1>, <601, 1>, <829,
1>, <5167, 1>, <6597973, 1> ]
```

However, z is not primitive, which can also be seen from the factorization of $5^{36} - 1$.

```
> IsPrimitive(z);
false
> Factorization(5^36-1);
[ <2, 4>, <3, 3>, <7, 1>, <13, 1>, <19, 1>, <31, 1>, <37, 1>, <601, 1>, <829,
1>, <5167, 1>, <6597973, 1> ]
```

Clearly z is the 8-th power of a primitive element. Let y be the 8-th root of z. Then y is primitive.

```
> (#F36 - 1) div Order(z);
8
> y := Root(z, 8);
> IsPrimitive(y);
true
> y;
w36^35 + 3*w36^33 + 4*w36^32 + w36^31 + 3*w36^30 + 2*w36^28 + w36^26 + 2*w36^25
    + 2*w36^23 + 4*w36^20 + 2*w36^19 + 2*w36^15 + 2*w36^14 + 4*w36^12 + 4*w36^11
    + w36^10 + 3*w36^9 + 4*w36^8 + 2*w36^7 + w36^6 + 4*w36^5 + 3*w36^3 + w36^2 +
    4*w36 + 2
> 5^36-1;
145519152283668518066406 24
```

Finally, we test whether y is normal over $F4$, i.e. whether $[y, y^q, y^{q^2}, y^{q^3}, \ldots]$ are independent over $F4$ where $q = \#F4 = 5^4$.

```
> IsNormal(y, F4);
true
```

Chapter 5

Number Fields

5.1 Features

- Simple and relative extensions
- Maximal order, integral basis (Round 2 and Round 4 algorithms)
- Quotient orders, suborders, extension orders
- Construction of integral and fractional ideals
- Ideal arithmetic: product, quotient, gcd, lcm
- Determination of whether an ideal is: integral, prime, principal
- Factorization of an ideal; Decomposition of primes
- Residue field of an order modulo a prime ideal
- Class group, ray class group
- Unit group, exceptional units, S-units, regulator
- Solution of: norm equations, relative norm equations
- Solution of: Thue equations, unit equations, index form equations, Mordell equations
- Determination of subfields
- Automorphism group of a normal extension
- Isomorphism of number fields
- Galois group of a polynomial (for degrees less than 12)

5.1.1 Example: Imprimitive degree 9 fields

In this example we illustrate how to construct some of the fields that are the subject of study in a certain paper, and show how to verify some of the results mentioned there. The paper is: F. Diaz y Diaz, M. Olivier, Imprimitive ninth-degree number fields with small discriminants, *Math. Comp.* **64** (1995), 305–321.

We begin with two special fields of signature $(9, 0)$: first the maximal real subfield of the cyclotomic field $\mathbb{Q}(\zeta_{19})$.

```
> R<x> := PolynomialRing(IntegerRing());
> C<c> := CyclotomicField(19);
> f := MinimalPolynomial(c + c^-1);
> M<m> := NumberField(f);
> Signature(M);
9 0
> M;
Number Field with defining polynomial x^9 + x^8 - 8*x^7 - 7*x^6 + 21*x^5 +
    15*x^4 - 20*x^3 - 10*x^2 + 5*x + 1 over the Rational Field
> Factorization(Discriminant(M));
[ <19, 8> ]
```

As expected, only the prime 19 ramifies in the field, and indeed it is totally ramified:

```
> Decomposition(MaximalOrder(M), 19);
[
    <Ideal of Equation Order of M
    Two element generators:
        [19, 0, 0, 0, 0, 0, 0, 0, 0]
        [17, 1, 0, 0, 0, 0, 0, 0, 0], 9>
]
```

The next field is the composite field of $k = \mathbb{Q}(\alpha)$ where $\alpha^3 = -\alpha^2 + 2\alpha + 1$, and $l = \mathbb{Q}(\gamma)$ where $\gamma^3 = 3\gamma + 1$. Note that we create a relative extension by a *rational* polynomial. The field contains two other non-isomorphic cubic subfields.

```
> k<a> := NumberField(x^3 + x^2 - 2*x - 1);
> Discriminant(k);
49

> l<b> := NumberField( x^3 - 3*x - 1 );
> CF := CompositeFields(k, l);
> N := CF[1];
> N;
Number Field with defining polynomial x^9 + 3*x^8 - 12*x^7 - 38*x^6 + 21*x^5 +
    93*x^4 - x^3 - 51*x^2 - 9*x + 1 over the Rational Field
> Factorization(Discriminant(N));
[ <3, 12>, <7, 6> ]
> S := Subfields(N);
> [ Discriminant(S[i][1]) : i in [1..#S]];
[ 62523502209, 3969, 3969, 81, 49 ]
> IsIsomorphic(S[2][1], S[3][1]);
false
```

The field L is one of 3 imprimitive degree 9 fields given in Diaz y Diaz and Olivier for which the class group is $C_3 \times C_3$. We find its class group. We also find the cubic subfield; we show that it is isomorphic to k above and create the explicit isomorphism.

```
> L := NumberField(x^6 - 2*x^5 + 20*x^4 - 27*x^3 + 140*x^2 - 98*x + 343);
> MinkowskiBound(L);
205
```

```
> C, m := ClassGroup(L);
> C;
Abelian Group isomorphic to Z/3 + Z/3
Defined on 2 generators
Relations:
    3*C.1 = 0
    3*C.2 = 0
> Norm( m(C.1) ), Norm( m(C.2) );
7 27
> S := Subfields(L);
> S[2][1];
Number Field with defining polynomial x^3 - 22*x^2 + 159*x - 377
over the Rational Field
> H<h> := S[2][1]; g := S[2][2];
> fl, f := IsIsomorphic(k, H);
> fl;
true
> f(k.1);
h^2 - 14*h + 47
> m<y> := MinimalPolynomial(g(f(k.1))); m;
y^3 + y^2 - 2*y - 1
```

In Diaz y Diaz and Olivier's paper, 4 non-isomorphic fields are constructed that are all relative cubic extensions of discriminant -2045563163 of the cubic field k of discriminant 49 defined above.

We construct the 4 fields, using the given cubic polynomials, as relative and absolute extensions. We check the discriminants; note that the computation of the discriminant triggers the determination of the ring of integers.

We invoke the `IsIsomorphic` function on a pair of the degree 9 fields.

By just looking at the decomposition of the primes $11, 17$ and 19 (as suggested by the authors) we see that the fields are non-isomorphic.

```
> R<x> := PolynomialRing(IntegerRing());
> k<a> := NumberField(x^3 + x^2 - 2*x - 1);
> b := a^2;
> Discriminant(k);
49

> S<s> := PolynomialRing(k);
> K1<z1> := ext< k | s^3 - (1-a)*s^2 + (4+a)*s - (5-a-2*b) >;
> L1 := AbsoluteField(K1);
> L1, Discriminant(L1);

Number Field with defining polynomial x^9 - 4*x^8 + 14*x^7 - 39*x^6 + 72*x^5 -
    110*x^4 + 128*x^3 - 78*x^2 + 16*x - 1 over the Rational Field
-2045563163

> K2<z2> := ext< k | s^3 - (1-a)*s^2 + (-5+a+3*b)*s - (-8+a+4*b) >;
> L2 := AbsoluteField(K2);
> L2, Discriminant(L2);
Number Field with defining polynomial x^9 - 4*x^8 + 2*x^7 + 11*x^6 - 28*x^5 +
```

```
    44*x^4 - 36*x^3 + 12*x^2 - 2*x - 13 over the Rational Field
-2045563163

> K3<z3> := ext< k | s^3 - (1-2*a-b)*s^2 + (-1+2*a+b)*s - (3-2*a-b) >;
> L3 := AbsoluteField(K3);
> L3, Discriminant(L3);
Number Field with defining polynomial x^9 - 7*x^7 - 13*x^6 + 42*x^4 + 82*x^3 +
    84*x^2 + 49*x + 13 over the Rational Field
-2045563163

> K4<z4> := ext< k | s^3 - (1-2*a-b)*s^2 + (-1+2*a+b)*s - (2-2*b) >;
> L4 := AbsoluteField(K4);
> L4, Discriminant(L4);
Number Field with defining polynomial x^9 - 7*x^7 - 3*x^6 + 14*x^5 + 7*x^4 -
    25*x^3 - 42*x^2 - 28*x - 8 over the Rational Field
-2045563163

> IsIsomorphic(L1, L2);
false
> [ [#Decomposition( MaximalOrder(L), p ) :
    p in [11, 13, 17] ] : L in [L1, L2, L3, L4] ];
[
    [ 3, 4, 1 ],
    [ 1, 6, 3 ],
    [ 1, 6, 1 ],
    [ 1, 4, 1 ]
]
```

5.1.2 Example: Galois Group and its Action

If it is possible to obtain the full factorization of an integer polynomial over its splitting field, the Galois action of the group of the splitting field can be made entirely explicit. In this example we show how it can be done in Magma, and how to find the Galois correspondence (between subgroups and subfields). Although there exists a polynomial-time algorithm for the factorization of polynomials over number fields, in practice this is the bottleneck for our approach to Galois theory. Only in small examples we will be able to construct the splitting field as below.

We begin with a cubic polynomial f and determine its Galois group using the intrinsic function `GaloisGroup(f)`. This will be a degree-3 representation.

```
> R<x> := PolynomialRing(RationalField());
> f := x^3 - x - 1;
> GaloisGroup(f);
Permutation group G acting on a set of cardinality 3
Order = 6 = 2 * 3
    (1, 2)
    (1, 2, 3)
```

Next, we find the Galois group in two degree-6 representations. Firstly, we construct it bare-handed. We start by obtaining the splitting field for f as a two-step extension of $\mathbb{Q}$.

```
> N<n> := NumberField(f);
> ff := Factorization( PolynomialRing(N) ! f );
> ff;
[
    <$.1 - n, 1>,
    <$.1^2 + n*$.1 + n^2 - 1, 1>
]
> M<m> := ext< N | ff[2][1] >;
> A<a> := AbsoluteField(M);
> A;
Number Field with defining polynomial x^6 - 6*x^4 + 9*x^2 + 23
over the Rational Field
```

We factorize f over the splitting field, and obtain all its roots from the linear factors.

```
> S<s> := PolynomialRing(A);
> factn := Factorization( S ! DefiningPolynomial(A) );
> factn;
[
    <s - a, 1>,
    <s + a, 1>,
    <s - 1/6*a^4 + 5/6*a^2 - 1/2*a - 2/3, 1>,
    <s - 1/6*a^4 + 5/6*a^2 + 1/2*a - 2/3, 1>,
    <s + 1/6*a^4 - 5/6*a^2 - 1/2*a + 2/3, 1>,
    <s + 1/6*a^4 - 5/6*a^2 + 1/2*a + 2/3, 1>
]
> C := [ -Coefficient(factr[1], 0) : factr in factn];
> C;
[
    a,
    -a,
    1/6*a^4 - 5/6*a^2 + 1/2*a + 2/3,
    1/6*a^4 - 5/6*a^2 - 1/2*a + 2/3,
    -1/6*a^4 + 5/6*a^2 + 1/2*a - 2/3,
    -1/6*a^4 + 5/6*a^2 - 1/2*a - 2/3
]
```

Each of the roots is an algebraic conjugate of the primitive element a of the field. The elements of the Galois group of A over $\mathbb{Q}$ are obtained by sending a to one of its conjugates. Thus the action on A of such an element of the Galois group is determined by the polynomial (with rational coefficients) expressing the conjugate in a. These polynomials are stored in P below. We (again) determine the Galois group: by numbering the algebraic conjugates c_i and finding all images of c_i under a given P_j, we obtain the permutation associated with P_j. The Galois group H consists of these permutations on 6 letters.

```
> P := [ R ! Eltseq(x) : x in C];
> P;
[
    x,
    -x,
    1/6*x^4 - 5/6*x^2 + 1/2*x + 2/3,
    1/6*x^4 - 5/6*x^2 - 1/2*x + 2/3,
```

```
    -1/6*x^4 + 5/6*x^2 + 1/2*x - 2/3,
    -1/6*x^4 + 5/6*x^2 - 1/2*x - 2/3
]
> I := [ [ Index(C, Evaluate(p, c)) : p in P ] : c in C];
> I;
[
    [ 1, 2, 3, 4, 5, 6 ],
    [ 2, 1, 4, 3, 6, 5 ],
    [ 3, 6, 1, 5, 4, 2 ],
    [ 4, 5, 2, 6, 3, 1 ],
    [ 5, 4, 6, 2, 1, 3 ],
    [ 6, 3, 5, 1, 2, 4 ]
]
> H := sub< Sym(6) | I >;
> H;
Permutation group H acting on a set of cardinality 6
    (1, 2)(3, 4)(5, 6)
    (1, 3)(2, 6)(4, 5)
    (1, 4, 6)(2, 5, 3)
    (1, 5)(2, 4)(3, 6)
    (1, 6, 4)(2, 3, 5)
```

Lastly, we find the Galois group G using the intrinsic function once more, but in terms of the degree-6 defining polynomial of the splitting field. It will be conjugate to H: a simple renumbering of roots makes them equal. As we see, we only need to cyclically permute three roots.

```
> G := GaloisGroup(DefiningPolynomial(A));
> fl, el := IsConjugate(Sym(6), G, H);
> fl, el;
true (3, 4, 5)
```

Since we have the explicit action of the Galois group, we can now find the quadratic subfield of A corresponding to the subgroup K of H of order 3. We create the sequence of automorphisms of A contained in K and see that the trace of a^3 generates the required quadratic field.

```
> S := Subgroups(H);
> 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
    (1, 2)(3, 4)(5, 6)
[3]     Order 3            Length 1
Permutation group acting on a set of cardinality 6
    (1, 6, 4)(2, 3, 5)
[4]     Order 6            Length 1
Permutation group acting on a set of cardinality 6
```

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

> K := S[3]`subgroup; // extract subgroup from the record S[3]
> J := [ hom< A -> A | C[1^k] > : k in K ];
> tra := &+[ h(C[1]^3) : h in J ];
> tra, MinimalPolynomial(tra);
3*a^3 - 9*a
x^2 + 207
> SquareFree(-207);
-23 3
```

Therefore the quadratic subfield is $\mathbb{Q}(\sqrt{-207}) = \mathbb{Q}(\sqrt{-23})$.

Chapter 6

Multivariate Polynomial Rings

6.1 Introduction

Multivariate polynomial rings in any number of variables may be formed over any coefficient ring, including a polynomial ring. Multivariate polynomials are represented in distributive form, using ordered linked lists of coefficient-monomial pairs. Different orderings are allowed on the monomials; these become significant in the construction of Gröbner bases of ideals. Computations with ideals are currently allowable over fields only.

6.2 Polynomial Rings: Creation and Ring Operations

- Creation of a polynomial ring with specific monomial order
- Monomial orders: lexicographical, graded lexicographical, graded reverse lexicographical, block elimination, general weight vectors
- Definition of a ring map
- Kernel of a ring map
- Properties of ring maps: Surjective, bijective

6.2.1 Example: Creation and Orders

This examples shows how one can construct different polynomial rings with different orders.

```
> Z := IntegerRing();
> // Construct polynomial ring with block elimination and a > d > b > c
> P<a,b,c,d> := PolynomialRing(Z, 4, "elim", [1, 4], [2, 3]);
> a + b + c + d;
a + d + b + c
> a + d^10 + b + c^10;
d^10 + a + c^10 + b
> a + d^10 + b + c;
d^10 + a + b + c

> // Construct polynomial ring with weight order and x > y > z
> P<x,y,z> := PolynomialRing(Z, 3, "weight", [100,10,1, 1,10,100, 1,1,1]);
```

```
> x + y + z;
x + y + z
> (x+y^2+z^3)^4;
x^4 + 4*x^3*y^2 + 4*x^3*z^3 + 6*x^2*y^4 + 12*x^2*y^2*z^3 + 6*x^2*z^6 + 4*x*y^6 +
    12*x*y^4*z^3 + 12*x*y^2*z^6 + 4*x*z^9 + y^8 + 4*y^6*z^3 + 6*y^4*z^6 +
    4*y^2*z^9 + z^12
```

6.3 Polynomial Rings: Arithmetic with Polynomials

- Monomial orders: lexicographical, graded lexicographical, graded reverse lexicographical, block elimination, general weight vectors
- Arithmetic with elements
- Recursive coefficient, monomial, term, and degree access
- Determination of whether an element is: a unit, a zero-divisor, nilpotent
- Differentiation, integration, evaluation and interpolation
- S-polynomial of two polynomials

6.3.1 Example: Interpolation

We let $P = \mathbb{Q}[x, y, z]$, and give an example of interpolation. We find a polynomial which, when evaluated in the first variable x at the rational points $1, 2, 3$, yields $y, z, y + z$ respectively. We check the result by evaluating.

```
> Q := RationalField();
> P<x, y, z> := PolynomialRing(Q, 3);
> f := Interpolation([Q | 1, 2, 3], [y, z, y + z], 1);
> f;
x^2*y - 1/2*x^2*z - 4*x*y + 5/2*x*z + 4*y - 2*z
> [ Evaluate(f, 1, v) : v in [1, 2, 3] ];
[
    y,
    z,
    y + z
]
```

6.4 Polynomial Rings: GCD and Resultant

- Resultant (modular and sub-resultant algorithms), discriminant
- Greatest common divisor (sparse EEZ-GCD and fast GCD-HEU algorithms over $\mathbb{Z}$ and $\mathbb{Q}$ and fast interpolation algorithms over GF(q) and fields of characteristic 0)

6.4.1 Example: Resultants

We illustrate in this example how resultants can be used to perform various operations in algebraic number fields.

Let K be the number field $\mathbb{Q}(\alpha)$ where $\alpha^{12} + \alpha + 1 = 0$ and let L be the extension field $K(\beta)$ of K where $\beta^5 - \beta^3 - \alpha = 0$. For this example we let f and g be the corresponding polynomials in $\mathbb{Q}[a, b]$.

```
> P<a,b> := PolynomialRing(RationalField(), 2);
> f := a^12 + a + 1;
> g := b^5 - b^3 - a;
```

Suppose we wish to the compute the minimal polynomial of β over $\mathbb{Q}$. There are many ways to do this, but using resultants we only need compute $\mathrm{Res}_a(f, g)$; the squarefree part of this is the minimal polynomial we desire, which is the resultant itself in this example.

```
> time Resultant(f, g, a);
b^60 - 12*b^58 + 66*b^56 - 220*b^54 + 495*b^52 - 792*b^50 + 924*b^48 - 792*b^46
    + 495*b^44 - 220*b^42 + 66*b^40 - 12*b^38 + b^36 + b^5 - b^3 + 1
Time: 0.030
> SquareFreeFactorization($1);
[
    <b^60 - 12*b^58 + 66*b^56 - 220*b^54 + 495*b^52 - 792*b^50 + 924*b^48 -
        792*b^46 + 495*b^44 - 220*b^42 + 66*b^40 - 12*b^38 + b^36 + b^5 - b^3 +
        1, 1>
]
```

To the compute the minimal polynomial of $\alpha + \beta$ over $\mathbb{Q}$, we compute $\mathrm{Res}_a(f, h)$ where $h(a, b) = g(a, b - a)$.

```
> time Resultant(f, Evaluate(g, b, b - a), a);
b^60 - 12*b^58 + 66*b^56 - 220*b^54 + 495*b^52 - 792*b^50 + 5*b^49 + 929*b^48 +
    72*b^47 - 696*b^46 + 11759*b^45 + 18063*b^44 - 44440*b^43 - 47080*b^42 +
    419309*b^41 + 762729*b^40 - 1716396*b^39 - 2873894*b^38 + 5425737*b^37 +
    10385840*b^36 - 13258851*b^35 - 26904947*b^34 + 25189520*b^33 +
    54980662*b^32 - 40186674*b^31 - 86106427*b^30 + 66820655*b^29 +
    139147277*b^28 - 32688591*b^27 - 163933051*b^26 - 111597501*b^25 +
    15063002*b^24 + 49758590*b^23 - 68256681*b^22 + 175128899*b^21 +
    310117887*b^20 - 193726874*b^19 - 135269679*b^18 + 83867563*b^17 -
    71486444*b^16 + 40304793*b^15 + 90794636*b^14 - 5975719*b^13 + 21427983*b^12
    + 8397176*b^11 + 7883626*b^10 - 3151102*b^9 + 5844757*b^8 - 1654516*b^7 +
    844960*b^6 - 175328*b^5 + 53371*b^4 - 8814*b^3 + 1721*b^2 - 170*b + 13
Time: 0.059
```

Notice that the lexicographical Gröbner basis (with $a > b$) of the ideal generated by f and g contains f and the minimal polynomial of β (written in b).

```
> GroebnerBasis([f, g]);
[
    a - b^5 + b^3,
    b^60 - 12*b^58 + 66*b^56 - 220*b^54 + 495*b^52 - 792*b^50 + 924*b^48 -
        792*b^46 + 495*b^44 - 220*b^42 + 66*b^40 - 12*b^38 + b^36 + b^5 - b^3 +
        1
]
```

6.5 Factorization

- Factorization over GF(q), $\mathbb{Z}$, $\mathbb{Q}$ (EEZ algorithm), $\mathbb{Q}(\alpha)$ (Trager algorithm) and $R(x_1, \ldots, x_n)$ where R is any of the previous rings

6.5.1 Example: Trinomial Factorization

We create a polynomial f in the polynomial ring in three indeterminates over the ring of integers by multiplying together various trinomials. The resulting product f has 461 terms and total degree 15. We then factorize f to recover the trionomials.

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> f := &*[x^i+y^j+z^k: i,j,k in [1..2]];
> #Terms(f);
461
> TotalDegree(f);
15
> time Factorization(f);
[
    <x + y + z, 1>,
    <x + y + z^2, 1>,
    <x + y^2 + z, 1>,
    <x + y^2 + z^2, 1>,
    <x^2 + y + z, 1>,
    <x^2 + y + z^2, 1>,
    <x^2 + y^2 + z, 1>,
    <x^2 + y^2 + z^2, 1>
]
Time: 0.360
```

6.5.2 Example: Factorization over an algebraic number field

We factorize a polynomial lying in $K[x, y, z]$, where K is the cyclotomic field $\mathbb{Q}(\zeta_5)$ of order 5. Notice that the polynomial has rational coefficients and is irreducible over $\mathbb{Q}$ but factors non-trivially over K.

```
> K<zeta_5> := CyclotomicField(5);
> P<x,y,z> := PolynomialRing(K, 3);
> f :=
>   x^8 - x^7*z + 4*x^6*y + x^6*z^2 + 4*x^6*z - 5*x^6 - 3*x^5*y*z -
>     x^5*z^3 - 3*x^5*z^2 + 5*x^5*z + 6*x^4*y^2 + 2*x^4*y*z^2 +
>     12*x^4*y*z - 15*x^4*y + x^4*z^4 + 2*x^4*z^3 + x^4*z^2 - 15*x^4*z +
>     10*x^4 - 3*x^3*y^2*z - x^3*y*z^3 - 6*x^3*y*z^2 + 10*x^3*y*z -
>     x^3*z^4 + 2*x^3*z^3 + 10*x^3*z^2 - 10*x^3*z + 4*x^2*y^3 +
>     x^2*y^2*z^2 + 12*x^2*y^2*z - 15*x^2*y^2 + 2*x^2*y*z^3 +
>     7*x^2*y*z^2 - 30*x^2*y*z + 20*x^2*y + x^2*z^4 - x^2*z^3 -
>     5*x^2*z^2 + 20*x^2*z - 10*x^2 - x*y^3*z - 3*x*y^2*z^2 + 5*x*y^2*z
>     - 3*x*y*z^3 + 10*x*y*z^2 - 10*x*y*z - x*z^4 + 5*x*z^3 - 10*x*z^2 +
```

```
>     10*x*z + y^4 + 4*y^3*z - 5*y^3 + 6*y^2*z^2 - 15*y^2*z + 10*y^2 +
>     4*y*z^3 - 15*y*z^2 + 20*y*z - 10*y + z^4 - 5*z^3 + 10*z^2 - 10*z + 5;
> time Factorization(f);
[
    <x^2 + zeta_5*x*z + y + z + zeta_5 - 1, 1>,
    <x^2 + zeta_5^2*x*z + y + z + zeta_5^2 - 1, 1>,
    <x^2 + (-zeta_5^3 - zeta_5^2 - zeta_5 - 1)*x*z + y + z - zeta_5^3 -
        zeta_5^2 - zeta_5 - 2, 1>,
    <x^2 + zeta_5^3*x*z + y + z + zeta_5^3 - 1, 1>
]
Time: 12.890
> IsIrreducible(PolynomialRing(RationalField(), 3) ! f);
true
```

6.6 Polynomial Rings: Arithmetic with Ideals

- Construction of ideals and subrings
- Construction of quotient rings
- Gröbner bases of ideals, with specialized algorithms for different coefficient fields (fraction-free methods for the rational field and rational function fields)
- Gröbner Walk algorithm for converting the Gröbner basis of an ideal with respect to one order to a basis with respect to a different monomial order
- Normal form of a polynomial with respect to an ideal
- Reduction of ideal bases
- Arithmetic with ideals: Sum, product, intersection, colon ideal, saturation of an ideal, leading monomial ideal
- Elimination ideals
- Determination of whether a polynomial is in an ideal or its radical
- Properties of an ideal: Zero, principal, proper, zero-dimensional
- Extension and contraction of ideals
- Variable extension of ideals

Magma computes the `lex` order Gröbner basis for the Katsura-5 problem in 4.5 seconds and the `lex` order Gröbner basis for the cyclic 6-th roots problem in 3.9 seconds. Taking a much more difficult example, Magma computes the `grevlex` order Gröbner basis for the cyclic 7-th roots problem in 780 seconds and transforms to the `lex` order Gröbner basis (using the FGLM algorithm) in a further 8563 seconds.

6.6.1 Example: Cyclic-6 Roots Lexicographical Gröbner Basis

We compute the Gröbner basis of the "Cyclic-6" ideal with respect to the lexicographical order. The ideal is an ideal of the polynomial ring $\mathbb{Q}(x, y, z, t, u, v)$. We also note that the last polynomial in the Gröbner basis is univariate (since, in fact, the ideal is zero-dimensional and the monomial order is lexicographical) and observe that it has a nice factorization. Note especially that in this example, homogenizing at first and keeping the Gröbner basis reduced makes this computation very fast; without using these features (i.e., if the parameters `Homogenize := false` or `ReduceByNew := false` are given), the computation is much more expensive (takes hundreds of seconds on the same computer).

```
> Q := RationalField();
> P<x,y,z,t,u,v> := PolynomialRing(Q, 6);
> I := ideal<P |
>     x + y + z + t + u + v,
>     x*y + y*z + z*t + t*u + u*v + v*x,
>     x*y*z + y*z*t + z*t*u + t*u*v + u*v*x + v*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*v + t*u*v*x + u*v*x*y + v*x*y*z,
>     x*y*z*t*u + y*z*t*u*v + z*t*u*v*x + t*u*v*x*y + u*v*x*y*z + v*x*y*z*t,
>     x*y*z*t*u*v - 1>;
> time B := GroebnerBasis(I);
Time: 3.870
> #B;
17
> B[17];
v^48 - 2554*v^42 - 399710*v^36 - 499722*v^30 + 499722*v^18 + 399710*v^12 +
    2554*v^6 - 1
> Factorization(B[17]);
[
    <v - 1, 1>,
    <v + 1, 1>,
    <v^2 + 1, 1>,
    <v^2 - 4*v + 1, 1>,
    <v^2 - v + 1, 1>,
    <v^2 + v + 1, 1>,
    <v^2 + 4*v + 1, 1>,
    <v^4 - v^2 + 1, 1>,
    <v^4 - 4*v^3 + 15*v^2 - 4*v + 1, 1>,
    <v^4 + 4*v^3 + 15*v^2 + 4*v + 1, 1>,
    <v^8 + 4*v^6 - 6*v^4 + 4*v^2 + 1, 1>,
    <v^8 - 6*v^7 + 16*v^6 - 24*v^5 + 27*v^4 - 24*v^3 +
        16*v^2 - 6*v + 1, 1>,
    <v^8 + 6*v^7 + 16*v^6 + 24*v^5 + 27*v^4 + 24*v^3 +
        16*v^2 + 6*v + 1, 1>
]
```

6.7 Polynomial Rings: Invariants for Ideals

- Dimension and maximally independent sets
- Hilbert series and Hilbert polynomial
- Primary decomposition of an ideal
- Probabilistic prime decomposition of the radical of an ideal
- Radical of an ideal
- Computation of the variety of a zero-dimensional ideal
- Relation ideals (determination of algebraic relations between polynomials)
- Syzygy modules
- Construction of a minimal generating set of a polynomial subalgebra
- Computations with polynomial generators of a submodule over a subalgebra in a polynomial ring

6.7.1 Example: Primary Decomposition and Radical of an Ideal

We create a certain symmetrical ideal I of the multivariate polynomial ring $\mathbb{Q}[x, y, z]$. The monomial order is the lexicographical order. We then examine the structure of the ideal and its variety by computing the primary decomposition and radical.

We first create the ideal I and examine its Gröbner basis.

```
> PR<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<PR |
>     (x + y + z)^3 - 1,
>     x^4 + y^4 + z^4 - (x + y + z)^2,
>     x*y + y*z + x*z - x*y*z >;
> time Groebner(I);
Time: 0.100
> I;
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Dimension 0
Groebner basis:
[
    x + y + 855217/21665*z^16 + 4457294/64995*z^15 + 235743/3095*z^14 +
        5048128/21665*z^13 + 22596787/64995*z^12 - 20049839/21665*z^11 -
        21200979/43330*z^10 + 1752061/4333*z^9 + 30180029/86660*z^8 -
        5080366/21665*z^7 + 6056394/21665*z^6 - 307362/3095*z^5 -
        4204203/86660*z^4 + z - 1,
    y^4 + 2692007/64995*y*z^16 + 4262548/64995*y*z^15 + 1761757/21665*y*z^14 +
        16832308/64995*y*z^13 + 23739599/64995*y*z^12 - 20170833/21665*y*z^11 -
        8868323/43330*y*z^10 + 1301348/4333*y*z^9 + 8902473/86660*y*z^8 -
        13042009/43330*y*z^7 + 8326763/21665*y*z^6 - 9057681/43330*y*z^5 +
        566747/12380*y*z^4 + y*z^3 - y*z^2 + y*z - y + 432223/129990*z^16 +
        309192/21665*z^15 + 1173093/43330*z^14 + 3284986/64995*z^13 +
        626271/6190*z^12 + 2023943/43330*z^11 - 13310187/86660*z^10 -
        645625/4333*z^9 - 3129223/173320*z^8 + 1393769/86660*z^7 +
        264197/43330*z^6 + 4002661/86660*z^5 + 1764741/173320*z^4 - z^3 + z^2 -
        z,
    y^2*z - y^2 - 674008/64995*y*z^16 - 2072342/64995*y*z^15 -
        878158/21665*y*z^14 - 5375852/64995*y*z^13 - 10908076/64995*y*z^12 +
        3035222/21665*y*z^11 + 10381016/21665*y*z^10 - 114777/4333*y*z^9 -
        5569833/21665*y*z^8 - 817339/43330*y*z^7 + 1543701/43330*y*z^6 -
        4213161/43330*y*z^5 + 483561/6190*y*z^4 + y*z^2 - 2*y*z + y -
        1891643/64995*z^16 - 794984/21665*z^15 - 772043/21665*z^14 -
        9768532/64995*z^13 - 3896237/21665*z^12 + 17014617/21665*z^11 +
        438947/43330*z^10 - 1637284/4333*z^9 - 1128671/12380*z^8 +
        10978071/43330*z^7 - 1950927/6190*z^6 + 8516229/43330*z^5 -
        2565651/86660*z^4 - z^2 + z,
    z^17 + z^16 + z^15 + 5*z^14 + 5*z^13 - 28*z^12 + 15/2*z^11 + 21/2*z^10 -
        3/4*z^9 - 33/4*z^8 + 27/2*z^7 - 21/2*z^6 + 15/4*z^5 - 3/4*z^4
]
```

We next compute the radical R of I. Notice that R properly contains I (so I is not radical).

```
> time R := Radical(I);
Time: 0.400
> R;
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Dimension 0, Radical
Groebner basis:
[
    x + y - 22941/21665*z^13 - 182992/64995*z^12 - 78683/21665*z^11 -
        181464/21665*z^10 - 1033091/64995*z^9 + 332397/21665*z^8 +
        1443007/43330*z^7 - 32738/4333*z^6 - 187171/12380*z^5 + 285491/43330*z^4
        + 983/6190*z^3 - 287451/43330*z^2 + 579969/86660*z - 1,
    y^2 - 6478/9285*y*z^12 - 1972/4333*y*z^11 + 2064/21665*y*z^10 -
        138392/64995*y*z^9 - 1674/3095*y*z^8 + 549446/21665*y*z^7 -
        94867/21665*y*z^6 - 275932/21665*y*z^5 + 1457/6190*y*z^4 +
        175498/21665*y*z^3 - 425097/43330*y*z^2 + 162198/21665*y*z - y -
        1074499/194985*z^13 - 1806491/194985*z^12 - 264368/21665*z^11 -
        1021568/27855*z^10 - 10456543/194985*z^9 + 7472996/64995*z^8 +
        601733/18570*z^7 - 754501/25998*z^6 - 777817/86660*z^5 +
        1434893/37140*z^4 - 13054459/259980*z^3 + 6603799/259980*z^2 -
        385237/64995*z,
    y*z^13 + 2*y*z^12 + 3*y*z^11 + 8*y*z^10 + 13*y*z^9 - 15*y*z^8 - 15/2*y*z^7 +
        3*y*z^6 + 9/4*y*z^5 - 6*y*z^4 + 15/2*y*z^3 - 3*y*z^2 + 3/4*y*z,
    z^14 + z^13 + z^12 + 5*z^11 + 5*z^10 - 28*z^9 + 15/2*z^8 + 21/2*z^7 -
        3/4*z^6 - 33/4*z^5 + 27/2*z^4 - 21/2*z^3 + 15/4*z^2 - 3/4*z
]

> I subset R;
true
> R subset I;
false
> IsRadical(I);
false
```

We next compute the *primary decomposition* of I, which consists of a sequence Q of primary ideals whose intersection is I, and a sequence P of associated prime ideals corresponding to Q. The primary decomposition gives a nice way of understanding the structure of I, and also shows how the variety of I decomposes.

```
> time Q, P := PrimaryDecomposition(I);
> #Q, #P;
4 4
> Q;
[
    Ideal of Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
    Dimension 0, Non-radical, Primary, Non-prime
    Size of variety over algebraically closed field: 1
    Groebner basis:
    [
        x + y,
```

```
        y^4,
        z - 1
    ],
    Ideal of Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
    Dimension 0, Non-radical, Primary, Non-prime
    Size of variety over algebraically closed field: 1
    Groebner basis:
    [
        x + z,
        y - 1,
        z^4
    ],
    Ideal of Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
    Dimension 0, Non-radical, Primary, Non-prime
    Size of variety over algebraically closed field: 1
    Groebner basis:
    [
        x - 1,
        y + z,
        z^4
    ],
    Ideal of Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
    Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 24
    Groebner basis:
    [
        x + y + 61112/64995*z^11 + 9482/4333*z^10 + 74792/21665*z^9 +
            79192/9285*z^8 + 322716/21665*z^7 - 208232/21665*z^6 -
            230586/21665*z^5 + 5587/3095*z^4 + 84806/21665*z^3 -
            198367/43330*z^2 + 116852/21665*z - 2951/6190,
        y^2 + 61112/64995*y*z^11 + 9482/4333*y*z^10 + 74792/21665*y*z^9 +
            79192/9285*y*z^8 + 322716/21665*y*z^7 - 208232/21665*y*z^6 -
            230586/21665*y*z^5 + 5587/3095*y*z^4 + 84806/21665*y*z^3 -
            198367/43330*y*z^2 + 116852/21665*y*z - 2951/6190*y +
            53057/64995*z^11 + 27833/12999*z^10 + 257296/64995*z^9 +
            616304/64995*z^8 + 377451/21665*z^7 + 14838/21665*z^6 -
            79997/43330*z^5 + 70123/43330*z^4 + 143529/86660*z^3 -
            373089/86660*z^2 + 301193/86660*z - 114169/86660,
        z^12 + 2*z^11 + 3*z^10 + 8*z^9 + 13*z^8 - 15*z^7 - 15/2*z^6 + 3*z^5 +
            9/4*z^4 - 6*z^3 + 15/2*z^2 - 3*z + 3/4
    ]
]
> P[1 .. 3];
[
    Ideal of Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
    Dimension 0, Radical, Prime
```

```
    Size of variety over algebraically closed field: 1
    Groebner basis:
    [
        x,
        y,
        z - 1
    ],
    Ideal of Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
    Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 1
    Groebner basis:
    [
        x,
        y - 1,
        z
    ],
    Ideal of Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
    Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 1
    Groebner basis:
    [
        x - 1,
        y,
        z
    ]
]
> P[4] eq Q[4];
true
```

There are thus 4 families of solutions to the set of equations implied by I. The first 3 families yield the solutions $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$. These solutions were obvious from the original basis of the ideal I. The last family (the variety of $P[4]$) yields a set of 24 solutions, lying in an algebraic number field of degree 24.

6.7.2 Example: Relation Ideals

We construct an ideal I of the polynomial ring $\mathrm{GF}(2)[x, y, z]$, and discover that the ideal is the full polynomial ring. Suppose we then wish to write $1 \in I$ as an (algebraic) expression in terms of our original generators of the ideal. We use `RelationIdeal` to find that expression.

```
> P<x, y, z> := PolynomialRing(GF(2), 3, "grevlex");
> S := [(x + y + z)^2, (x^2 + y^2 + z^2)^3 + x + y + z + 1];
> I := ideal<P | S>;
> Groebner(I);
> I;
Ideal of Polynomial ring of rank 3 over GF(2)
Graded Reverse Lexicographical Order
```

```
Variables: x, y, z
Groebner basis:
[
    1
]
> Q<a, b> := PolynomialRing(GF(2), 2);
> R := RelationIdeal(S, Q);
> R;
Ideal of Polynomial ring of rank 2 over GF(2)
Lexicographical Order
Variables: a, b
Basis:
[
    a^6 + a + b^2 + 1
]
> // Check the expression:
> S[1]^6 + S[1] + S[2]^2;
1
```

6.8 Polynomial Rings: Gradings

- Construction of graded polynomial rings with specific weights on the variables
- All monomials of specific total or weighted degree
- Homogeneous components of polynomials
- Homogenization and dehomogenization of an ideal
- Hilbert-driven Buchberger algorithm for fast computation of the Gröbner basis of a homogeneous ideal when the Hilbert series is known

6.8.1 Example: Hilbert-driven Buchberger algorithm

We illustrate a subalgorithm of the Invariant Theory module of Magma which uses the Hilbert-driven Buchberger Algorithm.

Let R be the invariant ring of the (permutation) cyclic group G of order 4 over the field $K = \mathrm{GF}(2)$. Suppose we have a sequence L of 4 homogeneous invariants of degrees $1, 2, 2$, and 4 respectively. We wish to determine efficiently whether the polynomials of L constitute primary invariants for R. To check this, the ideal generated by L must be zero-dimensional and the elements of L must be algebraically independent. This is equivalent to the condition that the weighted numerator of the Hilbert series of the ideal is the product $(1-t)(1-t^2)^2(1-t^4)$. If that is not the correct weighted numerator, it will be less than the correct weighted numerator so the algorithm will return true if and only if the polynomials L do constitute primary invariants for R.

```
> K := GF(2);
> P<a,b,c,d> := PolynomialRing(K, 4);
> L := [
>     a + b + c + d,
>     a*b + a*d + b*c + c*d,
>     a*c + b*d,
>     a*b*c*d
```

```
> ];
> // Form potential Hilbert series weighted numerator
> T<t> := PolynomialRing(IntegerRing());
> N := &*[1 - t^TotalDegree(f): f in L];
> N;
t^9 - t^8 - 2*t^7 + 2*t^6 + 2*t^3 - 2*t^2 - t + 1
> time l, B := HilbertGroebnerBasis(L, N);
> l;
true
```

Since `HilbertGroebnerBasis` returns true, the elements of L are algebraically independent (which is a well-known fact anyway since these elements are the elementary symmetric polynomials). The sequence B also has the Gröbner basis of the ideal generated by L.

```
> // Examine Groebner basis B of L:
> B;
[
    a + b + c + d,
    b^2 + d^2,
    b*c + b*d + c^2 + c*d,
    c^3 + c^2*d + c*d^2 + d^3,
    d^4
]
```

6.9 Affine Algebras

Let K be a field, $R = K[x_1, \ldots, x_n]$ a polynomial ring over K and I an ideal of R. The quotient ring $A = R/I$ is called an *affine algebra*.

6.10 Affine Algebras: Creation and Operations

- Creation of an affine algebra
- Arithmetic with elements

6.10.1 Example: Minimal Polynomial of an Algebraic Number

Suppose we wish to find the minimal polynomial of the algebraic number $\theta = \sqrt{2} + \sqrt[3]{5}$ over $\mathbb{Q}$. One way to do is to just find the minimal polynomial of (the coset of) $x + y$ over $\mathbb{Q}$ in the affine algebra $Q[x, y]/(x^2 - 2, y^3 - 5)$. This is easily done as follows.

```
> P<x, y> := PolynomialRing(RationalField(), 2);
> Q<a, b> := quo<P | x^2 - 2, y^3 - 5>;
> UP<z> := PolynomialRing(RationalField());
> M := MinimalPolynomial(a + b);
> M;
z^6 - 6*z^4 - 10*z^3 + 12*z^2 - 60*z + 17
```

```
> Evaluate(M, a + b);
0
```

6.11 Modules over Affine Algebras

Modules over a multivariate polynomial ring $K[x_1, \ldots, x_n]$ (K a field) and quotient rings of such (affine algebras) form a special category in Magma. Multivariate polynomial rings are not principal ideal rings in general, so the standard matrix echelonization algorithms are not applicable. Magma allows computations in modules over such rings by adding a column field to each monomial of a polynomial and then by using the ideal machinery based on Gröbner bases. This method is much more efficient than introducing new variables to represent the columns since the number of columns does not affect the total number of variables.

6.12 Modules over Affine Algebras: Creation and Operations

- Construction of modules with `TOP` ("term over position") or `POT` ("position over term") module orders
- Construction of graded modules with weights on the columns (determining homogeneity)
- Arithmetic with elements
- Construction of Gröbner bases of modules
- Row and column operations on elements

6.12.1 Example: Constructing Modules

We construct simple generic free modules over $\mathbb{Q}[x, y, z]$. The first module has default weights 0 on its columns, while the second has weights 1, 2, and 3 respectively on its columns.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> M := Module(P, 3);
> M;
Full Module of degree 3
TOP Order
Coefficient ring:
    Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
> GM := Module(P, [1, 2, 3], "pot");
> GM;
Full Module of degree 3
POT Order
Column weights: 1 2 3
Coefficient ring:
    Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
```

We now construct a module over a quotient ring.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> Q<a, b, c> := quo<P | y^3 + z*x - 2>;
> M := Module(Q, 3);
> M;
Full Quotient Module of degree 3
TOP Order
Coefficient ring:
    Ideal of Quotient Ring by
    Ideal of Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
    Basis:
    [
        x*z + y^3 - 2
    ]
    Preimage ideal:
    Polynomial ring of rank 3 over Rational Field
    Lexicographical Order
    Variables: x, y, z
Quotient Relations:
(            0            0 x*z + y^3 - 2)
(            0 x*z + y^3 - 2            0)
(x*z + y^3 - 2            0            0)
```

6.13 Modules over Affine Algebras: Submodules

- Construction of submodules and quotient modules
- Membership testing
- Hilbert series of (homogeneous) modules
- Submodule sum, intersection, colon operation
- Minimal bases for homogeneous modules

6.13.1 Example: Hilbert Series of a Module

We construct the Hilbert series of a simple homogeneous module.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> M := Module(P, 3);
> S := sub<M | [x, 0, z], [x^5 + y^5, z^5, y^3*x^2]>;
> IsHomogeneous(S);
true
> H<t> := HilbertSeries(S);
> H;
1/(t^2 - 2*t + 1)
```

6.14 Modules over Affine Algebras: Homology

- Syzygy modules
- Free resolutions, minimal free resolutions

6.14.1 Example: FreeResolution

We construct the free resolution of a certain module.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3, "grevlex");
> M := Module(P, 3);
> B := [[x*y, x^2, z], [x*z^3, x^3, y], [y*z, z, x],
>       [z, y*z, x], [y, z, x]];
> S := sub<M | B>;
> F := MinimalFreeResolution(S);
> #F;
3
> #Basis(F[2]);
35
> #MinimalBasis(F[2]);
5
> #Basis(F[3]);
6
> #MinimalBasis(F[3]);
3
> MinimalBasis(F[3]);
[
    (-x*y*z + x*z^2 + x^2 + 2*x*y - 2*x   -3*y + z + 2   1   -y 0),
    (x*y*z - x*y + z^2   -x + y   x   -x   0),
    (x^2*y*z^2 - x^2*z^3 - 3*x^2*y*z + x^2*z^2 + 2*x^2*y + x*z -
        2*x   x^2*y + 3*x*y*z - x*z^2 - 3*x*y - 3*y + 3   -y
        x*y*z - x*y + 1   -y*z + y)
]
> HomologicalDimension(S);
2
```

Chapter 7

Function Fields

7.1 General Function Fields

Given any field K and indeterminates $x_1, \ldots, x_n$, the user may form the field of rational functions $K(x_1, \ldots, x_n)$ as the localization of the polynomial ring $K[x_1, \ldots, x_n]$ at the prime ideal $\langle x \rangle$. At present only arithmetic and elementary operations are available in general function fields.

7.2 Algebraic Function Fields

A function field is a finite algebraic extension $K(x, \alpha)$ of the field of rational functions $K(x)$, where K is either $\mathbf{F}_q$, $\mathbb{Q}$ or a number field.

- Construction as an extension of $K(x)$
- Arithmetic with elements
- Exact constant field
- Genus
- Places, divisors and Riemann-Roch spaces
- Finite and infinite equation orders
- Discriminant (of an order)
- Maximal order (finite or infinite)
- Arithmetic of fractional ideals
- Properties of ideals: Integral, prime, principal
- Factorization of an ideal as an intersection of prime ideals
- Ramification index and residue class degree of a prime ideal
- Basis reduction for finite orders (Pohst-Schörnig method) for global fields
- Independent units, fundamental units, regulator for global fields
- Determination of places lying over a (finite) prime or the infinite place

- Determination of places of degree one in global function fields
- Determination of subfields

The development of this module is a joint project with the KANT group. Work is under way on developing algorithms for computing the divisor class group.

7.2.1 Example: Invariants

We create the rational function field $\mathbf{F}_{16}(x)$ and extend it by a root of the polynomial $y^4 + y + x^5$.

```
> K<k> := GF(2, 4);
> X<x> := FunctionField(K);
> Y<y> := PolynomialRing(X);
> f := y^4+y+x^5;
> F<y> := FunctionField(f);
> F;
Global function field defined over
Finite field of size 2^4 by
y^4 + y + x^5
> Signature(F);
[ <4, 1> ]
DimensionOfExactConstantField(F);
1
> Genus(F);
6
```

Thus, the exact constant field is K. We now construct the equation order at infinity, the maximal order at infinity and finally the ideals corresponding to pairwise distinct places lying over the infinite place of F.

```
> E := EquationOrderInfinite(F);
> E;
Infinite order over Finite field of size 2^4 defined by
y^4 + 1/x^6*y + 1/x^3

> Oinf := MaximalOrderInfinite(F);
> Oinf;
Infinite order over Finite field of size 2^4 defined by
y^4 + 1/x^6*y + 1/x^3
 and transformation matrix
[1/x^2     0     0     0]
[    0 1/x^2     0     0]
[    0     0   1/x     0]
[    0     0     0     1]
den: 1/x^2

> Decomposition(Oinf);
[ Ideal of Oinf
Basis:
```

```
[ 1/x    0    0    0]
[   0    1    0    0]
[   0    0    1    0]
[   0    0    0    1] ]
```

Thus there is only one ideal lying over the infinite place.

Chapter 8

Algebraically Closed Fields

8.1 Introduction

Algebraically Closed Fields are finite algebraic extensions of $\mathbb{Q}$ which automatically extend whenever there is a need. The technique is similar to the D5 method of D. Duval et al.

Any function in Magma which works generically over a field works automatically over these fields (including matrix and polynomial algorithms). The computation of roots and the factorization of polynomials over these fields also work automatically (such polynomials always split into linear factors of course). The Gröbner basis machinery works well over these fields also, allowing the computation of the full algebraic variety of ideals.

Factorization is delayed and there is an elaborate system whereby if a factor is found, the field automatically changes itself and reduces its elements to take advantage of the new information.

8.2 Algebraically Closed Fields: Creation and Operations

- Creation
- Minimal Polynomials
- Conjugates
- Roots and factorization of polynomials
- Simplification of conjugates
- Computation of absolute field
- Conversion to standard field

8.2.1 Example: Complete Jordan Form of a matrix over C

We compute the complete Jordan form of an 8×8 matrix with rational entries. The resulting canonical form has entries in the algebraic closure $\mathbf{A}$ of $\mathbb{Q}$.

First we create the matrix X with coefficient ring $\mathbf{A}$.

```
> A := AlgebraicClosure();
> P<x> := PolynomialRing(A);
```

```
> M := MatrixRing(A, 8);
> X := M !
> [ 0,0,1,0,0,0,0,0, -1,1,1,0,-1,0,-1,1,
>     1,0,1,0,0,0,0,0, 0,0,-1,1,-1,1,0,0,
>     0,0,1,0,-1,0,-2,1, 0,1,1,-2,0,-1,2,0,
>     0,1,-1,0,0,0,2,-1, -1,2,0,0,-1,0,0,0 ];

> X;
[ 0  0  1  0  0  0  0  0]
[-1  1  1  0 -1  0 -1  1]
[ 1  0  1  0  0  0  0  0]
[ 0  0 -1  1 -1  1  0  0]
[ 0  0  1  0 -1  0 -2  1]
[ 0  1  1 -2  0 -1  2  0]
[ 0  1 -1  0  0  0  2 -1]
[-1  2  0  0 -1  0  0  0]
```

Next we compute the Jordan form J of X, the transformation matrix T such that $TXT^{-1} = J$, and the primary invariant factors F of X. Notice that J is a true Jordan form with linear Jordan blocks.

```
> time J, T, F := JordanForm(X);
Time: 0.060
> J
[     r1       0       0       0       0       0       0       0]
[      0      r1       1       0       0       0       0       0]
[      0       0      r1       0       0       0       0       0]
[      0       0       0 -r1 + 1       0       0       0       0]
[      0       0       0       0 -r1 + 1       1       0       0]
[      0       0       0       0       0 -r1 + 1       0       0]
[      0       0       0       0       0       0      r3       0]
[      0       0       0       0       0       0       0     -r3]
```

We now print out A which shows the algebraic relations (minimal polynomials) which the indeterminates satisfy.

```
> A;
Algebraically closed field with 4 indeterminates
Relations:
[
    r4 + r3,
    r3^2 + 1,
    r2 + r1 - 1,
    r1^2 - r1 - 1
]
```

Finally we print T and F and verify that T transforms X to J correctly.

```
> T;
[1   -r1   -1   0   1   0   1   -1]
```

```
[1   -20/19*r1 + 15/19   28/19*r1 - 2/19   0   0   0   -25/19*r1 - 5/19
    20/19*r1 - 15/19]
[9/19*r1 - 2/19   0   7/19*r1 + 9/19   0   0   0   0   0]
[1   r1 - 1   -1   0   1   0   1   -1]
[1   20/19*r1 - 5/19   -28/19*r1 + 26/19   0   0   0   25/19*r1 - 30/19
    -20/19*r1 + 5/19]
[-9/19*r1 + 7/19   0   -7/19*r1 + 16/19   0   0   0   0   0]
[0   -4*r3 - 3   0   7*r3 - 1   -4*r3 - 3   4*r3 + 3   -11*r3 - 2   4*r3 + 3]
[0   4*r3 - 3   0   -7*r3 - 1   4*r3 - 3   -4*r3 + 3   11*r3 - 2   -4*r3 + 3]

> T*X*T^-1 eq J;
true


> F;
[
    <x - r1, 1>,
    <x - r1, 2>,
    <x + r1 - 1, 1>,
    <x + r1 - 1, 2>,
    <x - r3, 1>,
    <x + r3, 1>
]
```

8.3 Algebraically Closed Fields: Varieties

- Computation of varieties of polynomial ideals

8.3.1 Example: Cyclic-5 Roots Variety

In this example we compute the full algebraic variety of the Cyclic-5 Roots ideal. We first note the Gröbner basis of the ideal I.

```
> Q := RationalField();
> P<x,y,z,t,u> := PolynomialRing(Q, 5);
> I := ideal<P |
>     x + y + z + t + u,
>     x*y + y*z + z*t + t*u + u*x,
>     x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y +
>         u*x*y*z, x*y*z*t*u - 1>;

> GroebnerBasis(I);
[
    x + y + z + t + u,
    y^2 + 3*y*u + 2*t^6*u + 6*t^5*u^2 + t^4*u^3 - 2*t^3*u^4 + t^2 -
        566/275*t*u^11 - 6273/25*t*u^6 + 69019/275*t*u - 1467/275*u^12 -
        16271/25*u^7 + 179073/275*u^2,
    y*z - y*u + z^2 + 2*z*u - 6/5*t^6*u - 19/5*t^5*u^2 - t^4*u^3 + t^3*u^4 -
```

```
        2*t^2 + 334/275*t*u^11 + 3702/25*t*u^6 - 40726/275*t*u + 867/275*u^12 +
        9616/25*u^7 - 105873/275*u^2,
    y*t - y*u - 2/5*t^6*u - 8/5*t^5*u^2 - t^4*u^3 + t^3*u^4 + 124/275*t*u^11 +
        1372/25*t*u^6 - 15106/275*t*u + 346/275*u^12 + 3838/25*u^7 -
        42124/275*u^2,
    y*u^5 - y + 1/55*u^11 + 13/5*u^6 - 144/55*u,
    z^3 + 2*z^2*u - 2*z*u^2 + t^6*u^2 + 2*t^5*u^3 - 2*t^4*u^4 + 2*t^2*u -
        232/275*t*u^12 - 2576/25*t*u^7 + 28018/275*t*u^2 - 568/275*u^13 -
        6299/25*u^8 + 69307/275*u^3,
    z*t - z*u + 8/5*t^6*u + 22/5*t^5*u^2 - t^3*u^4 + t^2 - 442/275*t*u^11 -
        4901/25*t*u^6 + 53913/275*t*u - 1121/275*u^12 - 12433/25*u^7 +
        136674/275*u^2,
    z*u^5 - z + 1/55*u^11 + 13/5*u^6 - 144/55*u,
    t^7 + 3*t^6*u + t^5*u^2 - t^2 - 398/55*t*u^11 - 4414/5*t*u^6 + 48787/55*t*u
        - 1042/55*u^12 - 11556/5*u^7 + 128103/55*u^2,
    t^2*u^5 - t^2 - 2/55*t*u^11 - 21/5*t*u^6 + 233/55*t*u - 8/55*u^12 - 89/5*u^7
        + 987/55*u^2,
    u^15 + 122*u^10 - 122*u^5 - 1
]
```

Now we compute the variety of I over the algebraic closure $\mathbf{A}$ of $\mathbb{Q}$. Each 5-tuple represents a solution to the original set of polynomials. There are 70 solutions in all.

```
> A := AlgebraicClosure();
> time v := Variety(I, A);
Time: 1.529

> #v;
70

> v;
[
    <1, 1, 1, -r1 - 3, r1>,
    <1, 1, 1, r1, -r1 - 3>,
    <1, 1, -r1 - 3, r1, 1>,
    <1, 1, r1, -r1 - 3, 1>,
    <1, -r1 - 3, r1, 1, 1>,
    <1, r1, -r1 - 3, 1, 1>,
    <1, r3, r3^2, r3^3, -r3^3 - r3^2 - r3 - 1>,
    <1, -r3^3 - r3^2 - r3 - 1, r3^3, r3^2, r3>,
    <1, r3^3, r3, -r3^3 - r3^2 - r3 - 1, r3^2>,
    <1, r3^2, -r3^3 - r3^2 - r3 - 1, r3, r3^3>,
    <-r1 - 3, 1, 1, 1, r1>,
    <-r1 - 3, r1, 1, 1, 1>,
    <r1, 1, 1, 1, -r1 - 3>,
    <r1, -r1 - 3, 1, 1, 1>,
    <r3, r3, -r3^2 - 2*r3 - 1, r3^2 - r3 + 1, r3>,
    <r3, r3, r3^2 - r3 + 1, -r3^2 - 2*r3 - 1, r3>,
    <r3, -r3^2 - 2*r3 - 1, r3^2 - r3 + 1, r3, r3>,
    <r3, r3^2, r3^3, -r3^3 - r3^2 - r3 - 1, 1>,
    <r3, r3^2 - r3 + 1, -r3^2 - 2*r3 - 1, r3, r3>,
    <r3, r3^3, 1, r3^2, -r3^3 - r3^2 - r3 - 1>,
    <-r3^2 - 2*r3 - 1, r3, r3, r3, r3^2 - r3 + 1>,
```

```
<-r3^2 - 2*r3 - 1, r3^2 - r3 + 1, r3, r3, r3>,
<r3, 1, -r3^3 - r3^2 - r3 - 1, r3^3, r3^2>,
<r3, r3, r3, -r3^2 - 2*r3 - 1, r3^2 - r3 + 1>,
<r3, r3, r3, r3^2 - r3 + 1, -r3^2 - 2*r3 - 1>,
<r3, -r3^3 - r3^2 - r3 - 1, r3^2, 1, r3^3>,
<r3^2, 1, r3^3, r3, -r3^3 - r3^2 - r3 - 1>,
<r3^2, r3, 1, -r3^3 - r3^2 - r3 - 1, r3^3>,
<r3^2, r3^2, -r3^3 - 2*r3^2 - r3, r3^3 - r3^2 + r3, r3^2>,
<r3^2, r3^2, r3^3 - r3^2 + r3, -r3^3 - 2*r3^2 - r3, r3^2>,
<r3^2, -r3^3 - 2*r3^2 - r3, r3^3 - r3^2 + r3, r3^2, r3^2>,
<r3^2, -r3^3 - r3^2 - r3 - 1, r3, r3^3, 1>,
<r3^2, r3^3, -r3^3 - r3^2 - r3 - 1, 1, r3>,
<r3^2, r3^3 - r3^2 + r3, -r3^3 - 2*r3^2 - r3, r3^2, r3^2>,
<r3^2 - r3 + 1, r3, r3, r3, -r3^2 - 2*r3 - 1>,
<r3^2 - r3 + 1, -r3^2 - 2*r3 - 1, r3, r3, r3>,
<-2*r3^3 - r3 - 1, -r3^3 + r3 + 1, r3^3, r3^3, r3^3>,
<-r3^3 + r3 + 1, -2*r3^3 - r3 - 1, r3^3, r3^3, r3^3>,
<-r3^3 - 2*r3^2 - r3, r3^3 - r3^2 + r3, r3^2, r3^2, r3^2>,
<-r3^3 - r3^2 - r3 - 1, 1, r3, r3^2, r3^3>,
<-r3^3 - r3^2 - r3 - 1, r3, r3^3, 1, r3^2>,
<-r3^3 - r3^2 - r3 - 1, r3^2, 1, r3^3, r3>,
<-r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1, r3^3 +
    2*r3^2 + 2*r3 + 1, 2*r3^3 + r3^2 + r3 + 2>,
<-r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1, 2*r3^3
    + r3^2 + r3 + 2, r3^3 + 2*r3^2 + 2*r3 + 1>,
<-r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1, r3^3 + 2*r3^2 + 2*r3 + 1,
    2*r3^3 + r3^2 + r3 + 2, -r3^3 - r3^2 - r3 - 1>,
<-r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1, 2*r3^3 + r3^2 + r3 + 2, r3^3
    + 2*r3^2 + 2*r3 + 1, -r3^3 - r3^2 - r3 - 1>,
<-r3^3 - r3^2 - r3 - 1, r3^3, r3^2, r3, 1>,
<-r3^3 - r3^2 - r3 - 1, r3^3 + 2*r3^2 + 2*r3 + 1, 2*r3^3 + r3^2 + r3 + 2,
    -r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1>,
<-r3^3 - r3^2 - r3 - 1, 2*r3^3 + r3^2 + r3 + 2, r3^3 + 2*r3^2 + 2*r3 + 1,
    -r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1>,
<r3^3, 1, r3^2, -r3^3 - r3^2 - r3 - 1, r3>,
<r3^3, r3, -r3^3 - r3^2 - r3 - 1, r3^2, 1>,
<r3^3, r3^2, r3, 1, -r3^3 - r3^2 - r3 - 1>,
<r3^3, -2*r3^3 - r3 - 1, -r3^3 + r3 + 1, r3^3, r3^3>,
<r3^3, -r3^3 + r3 + 1, -2*r3^3 - r3 - 1, r3^3, r3^3>,
<r3^3, -r3^3 - r3^2 - r3 - 1, 1, r3, r3^2>,
<r3^3, r3^3, -2*r3^3 - r3 - 1, -r3^3 + r3 + 1, r3^3>,
<r3^3, r3^3, -r3^3 + r3 + 1, -2*r3^3 - r3 - 1, r3^3>,
<r3^3 - r3^2 + r3, -r3^3 - 2*r3^2 - r3, r3^2, r3^2, r3^2>,
<r3^3 + 2*r3^2 + 2*r3 + 1, -r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1,
    -r3^3 - r3^2 - r3 - 1, 2*r3^3 + r3^2 + r3 + 2>,
<r3^3 + 2*r3^2 + 2*r3 + 1, 2*r3^3 + r3^2 + r3 + 2, -r3^3 - r3^2 - r3 - 1,
    -r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1>,
<2*r3^3 + r3^2 + r3 + 2, -r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1, -r3^3
    - r3^2 - r3 - 1, r3^3 + 2*r3^2 + 2*r3 + 1>,
<2*r3^3 + r3^2 + r3 + 2, r3^3 + 2*r3^2 + 2*r3 + 1, -r3^3 - r3^2 - r3 - 1,
    -r3^3 - r3^2 - r3 - 1, -r3^3 - r3^2 - r3 - 1>,
<-r7 - r3^3 - r3^2 - 1, -r7*r3^3 - r7*r3^2 + r7 + r3^3 + r3^2, -r7*r3^3 -
    r7*r3^2 + r7 + r3^3 + r3^2, -r7*r3^3 - r7*r3^2 + r7 + r3^3 + r3^2,
    3*r7*r3^3 + 3*r7*r3^2 - 2*r7 - 2*r3^3 - 2*r3^2 + 1>,
```

```
    <r7, r7*r3^3 + r7*r3^2 - r7, r7*r3^3 + r7*r3^2 - r7, r7*r3^3 + r7*r3^2 - r7,
    -3*r7*r3^3 - 3*r7*r3^2 + 2*r7>,
    <-3*r7*r3^3 - 3*r7*r3^2 + 2*r7, r7*r3^3 + r7*r3^2 - r7, r7*r3^3 + r7*r3^2 -
        r7, r7*r3^3 + r7*r3^2 - r7, r7>,
    <-r7*r3^3 - r7*r3^2 + r7 + r3^3 + r3^2, -r7*r3^3 - r7*r3^2 + r7 + r3^3 +
        r3^2, -r7*r3^3 - r7*r3^2 + r7 + r3^3 + r3^2, -r7 - r3^3 - r3^2 - 1,
        3*r7*r3^3 + 3*r7*r3^2 - 2*r7 - 2*r3^3 - 2*r3^2 + 1>,
    <-r7*r3^3 - r7*r3^2 + r7 + r3^3 + r3^2, -r7*r3^3 - r7*r3^2 + r7 + r3^3 +
        r3^2, -r7*r3^3 - r7*r3^2 + r7 + r3^3 + r3^2, 3*r7*r3^3 + 3*r7*r3^2 -
        2*r7 - 2*r3^3 - 2*r3^2 + 1, -r7 - r3^3 - r3^2 - 1>,
    <r7*r3^3 + r7*r3^2 - r7, r7*r3^3 + r7*r3^2 - r7, r7*r3^3 + r7*r3^2 - r7, r7,
    -3*r7*r3^3 - 3*r7*r3^2 + 2*r7>,
    <r7*r3^3 + r7*r3^2 - r7, r7*r3^3 + r7*r3^2 - r7, r7*r3^3 + r7*r3^2 - r7,
        -3*r7*r3^3 - 3*r7*r3^2 + 2*r7, r7>,
    <3*r7*r3^3 + 3*r7*r3^2 - 2*r7 - 2*r3^3 - 2*r3^2 + 1, -r7*r3^3 - r7*r3^2 + r7
        + r3^3 + r3^2, -r7*r3^3 - r7*r3^2 + r7 + r3^3 + r3^2, -r7*r3^3 - r7*r3^2
        + r7 + r3^3 + r3^2, -r7 - r3^3 - r3^2 - 1>
]
```

We finally print out A so we can see the defining relations for `r1`, `r3` and `r7`.

Factorizations of intermediate defining polynomials have been found along the way so the other roots of the defining polynomials have been expressed in terms of `r1`, `r3` and `r7`.

```
> A;
Algebraic closure of Q
Defining relations:
[
    r10 - r3^3 - 2*r3^2 - 2*r3 - 1,
    r9 + r7 + r3^3 + r3^2 + 1,
    r8 + r3^2 + 2*r3 + 1,
    r7^2 + r7*r3^3 + r7*r3^2 + r7 +
        3*r3^3 + 3*r3^2 + 5,
    r6 - r3^3,
    r5 - r3^2,
    r4 + r3^3 + r3^2 + r3 + 1,
    r3^4 + r3^3 + r3^2 + r3 + 1,
    r2 + r1 + 3,
    r1^2 + 3*r1 + 1
]
```

Chapter 9

The Real and Complex Fields

9.1 Introduction

Two different models of the real and complex field are available in Magma. The default version is based on semantics developed by Henri Cohen for PARI. In this model, the precision of a real or complex number is determined by the accuracy of the operands and the operation. Whenever a real or complex number is known exactly, it is kept in exact form and only converted to real/complex form when it has to be used as an argument in some operation. Thus, during a calculation, real or complex numbers will appear with varying precisions, where the precision for a particular number is chosen in such a way that all digits should be meaningful. This is achieved through use of a form of interval arithmetic. We call this model of the real or complex field, the *free* model. Magma implements its free model using a modified version of the PARI code. The PARI code achieves its speed by using assembler for a small number of critical operations and by careful organization.

9.2 The Real and Complex Fields

- Arithmetic
- Square root, arithmetic-geometric mean
- Continued fraction expansion of a real number
- Constants: π, Euler's constant, Catalan's constant
- Logarithm, dilogarithm, exponential
- Trigonometric functions, hyperbolic functions and their inverses
- Bernoulli numbers
- Γ function, incomplete Γ function, complementary incomplete Γ function, logarithm of Γ function
- J-Bessel function, K-Bessel function
- U-confluent hypergeometric function
- Logarithmic integral, exponential integral
- Error function, complementary error function
- Log derivative (ψ) function, i.e, $\frac{\Gamma'(x)}{\Gamma(x)}$

- Riemann-ζ function
- Polylogarithm, Zagier's modifications of the polylogarithm
- Integer polynomial having a given real or complex number as an approximate root (Hastad, Lagarias and Schnorr LLL-method)
- Roots of an exact polynomial to a specified precision (Schönhage splitting circle method)
- Summation of a series (Euler-Wijngaarden method for alternating series)
- Numerical integration of a function (Romberg-type methods)

9.3 Elliptic and Modular Functions

The standard elliptic and modular functions are available in two forms. Firstly, their Fourier expansions may be calculated (i.e., power series in $q = e^{2\pi is}$). Secondly, their value may be obtained at any point in the upper-half of the complex plane.

- Dedekind η function
- Jacobi sine theta-function and its k-th derivative
- Modular j-function
- Weber's f-function, Weber's f_2-function
- Weierstrass ρ function

Chapter 10

Finitely Presented Groups

10.1 Introduction

Given an fp-group about which nothing is known, the immediate problems are to determine whether it is trivial, finite, infinite, free etc and to determine its finite homomorphic images, finite index subgroups and so on. The central strategy for analyzing an fp-group is to attempt to construct non-trivial homomorphisms, which may be onto an abelian group, p-group, nilpotent group, soluble group, permutation group (the Todd-Coxeter algorithm) or matrix group (vector enumeration).

10.2 Construction and Coset Enumeration

- Construction as a quotient of a free group
- Direct product, free product
- Arithmetic (free reduction only on words)
- Construction of subgroups and quotient groups
- Actions on coset spaces (Todd-Coxeter procedure);
- Actions on vector spaces (vector enumeration)
- Permutation representation on the cosets of a subgroup
- Maximal central extensions

10.2.1 Verifying Correctness of a Presentation for 3M(24)

The Todd-Coxeter procedure is the fundamental tool for investigating finitely presented groups. Magma contains the recent version developed by George Havas. The following example is a 19-generator presentation for the group 3M(24) constructed by Richard Weiss. The calculations presented below (made available by Gernot Stroth) were part of the process of verifying correctness.

```
> G<a,b,c,wa,wb,wc,h,d,e,w,z,f,ab,bc,bbc,abc,abbc,aabbc,u> :=
>   Group<a,b,c,wa,wb,wc,h,d,e,w,z,f,ab,bc,bbc,abc,abbc,aabbc,u|
>   ab*(a,b)^-1, bc*(b^wc)^-1, bbc*((b,c)*bc)^-1, abc*(bc^wa)^-1,
>   abbc*(abc^wb)^-1, aabbc*(bbc^wa)^-1, u*(z*w)^-1,
>   a^2, b^2, c^2, z^2, w^2, u^2, d^2, wa^2, wb^2, wc^2, e^2, f^2,
>   z*(a,b^h)*ab^(h^-1), w*(a,b^(h^-1))*ab^h,
```

```
>   (a,c), (a,ab), (b,bc), (c,bbc), (wa*wb)^3, (wa*a)^3, (wb*b)^3,
>   (wc,wa), (wb*wc)^4, (wc*c)^3, b^wa*ab,c^wa*c, abbc^wa*abbc,
>   a^wb*ab, c^wb*bbc, bc^wb*bc, aabbc^wb*aabbc, a^wc*a,bbc^wc*bbc,
>   abbc^wc*abbc, h^3,(h*b*b^wb)^5,h^wb*h,(b,bc^h)*bbc*u, (h,c),
>   h^wa*h, (h,wc), u*(b*h)^3, (w,a), (w,c), (z,a), (z,c), (w,wa),
>   (w,wb), (w,wc), (w,h), (w,z), (z,wa), (z,wb), (z,wc), (z,h),
>   (e,b)*z, (e,c)*z, (e,wc)*z, (e,wb)*z, (wa,e)*h^-1*z, h^e*h,
>   (d*e)^3*u*aabbc,
>   (wa*d)^3, (wb,d), (d,a), h^d*h, (wc,d)*aabbc, (b,d)*abbc*ab,
>   (f,a), (f,b), (f,c), (f,wa), (f,wb), (f,wc), (f,d), h^f*h, (e*f)^3 >;
>
> H := sub< G | a,b,c,bc,wa,wb,wc,h,z,w,d,e >;
```

Having set up a presentation for the group G and generators for a subgroup H, we try to construct the right coset space V for H in G.

```
> V := CosetSpace(G, H: CosetLimit := 2000000, Hard := true, Print := 1);
Fill Factor = 35, CT Factor = 1000, RT Factor = 1
Index = 920808, Max = 920808, Total = 921735
Time = 216.710 seconds
```

We were successful! The coset space has cardinality 920 808. From this we construct the permutation representation f onto P of G afforded by this coset space.

```
> f, P := CosetAction(V);
```

We use the random Schreier-Sims algorithm for a quick estimate of the order of the permutation group P.

```
> RandomSchreier(P);
> Order(P);
3765617127571985163878400
> FactoredOrder(P);
[ <2, 21>, <3, 17>, <5, 2>, <7, 3>, <11, 1>, <13, 1>, <17, 1>, <23, 1>,
<29, 1> ]
```

This is the correct order for 3M(24). If desired we could run a verification procedure and prove that this is the correct order for P.

10.3 Operations on Subgroups of Finite Index

A subgroup of a finitely presented group may be represented by its coset table. This representation allows us to compute various functions of the subgroup as shown in the following list:

- Membership,
- Core, intersection normalizer, normal closure
- Transversal
- Whether conjugate, maximal, normal, etc

10.3.1 Subgroup Calculations in a Space Group

We illustrate some of the subgroup constructions by applying them to construct subgroups of small index in the two-dimensional space group p4g which has presentation

$$< r, s | r^2, s^4, (r, s)^2 > .$$

```
> p4g<r, s> := Group< r, s | r^2 = s^4 = (r*s^-1*r*s)^2 = 1 >;
> p4g;
GrpFP: p4g on 2 generators
Relations
      r^2 = Id(p4g)
      s^4 = Id(p4g)
      (r * s^-1 * r * s)^2 = Id(p4g)
> h := sub< p4g | (s^-1 * r)^4, s * r >;
> k := sub< p4g | (s^-1 * r)^2, (s * r)^2 >;
> Index(p4g, h);
8
> Index(p4g, k);
8
> n := NormalClosure(p4g, h);
> n;
Finitely presented group n on 5 generators
Index in group p4g is 2
Generators as words in group p4g
      n.1 = s * r
      n.2 = (s^-1 * r)^4
      n.3 = r^-1 * s * r^2
      n.4 = r * s
      n.5 = s^2 * r * s^-1
> m := MinimalOvergroup(p4g, h);
> m;
Finitely presented group m on 3 generators
Index in group p4g is 4 = 2^2
Generators as words in group p4g
      m.1 = s * r
      m.2 = (s^-1 * r)^4
      m.3 = (r * s)^2
> l := h meet k;
> l;
Finitely presented group l
Subgroup of group p4g defined by coset table
> j := h^s;
> j;
Finitely presented group j on 2 generators
Generators as words in group p4g
      j.1 = r * s
      j.2 = (s^-2 * r * s)^4
> j eq h;
false
> IsConjugate(p4g, h, j);
true r^-1
> c := Core(p4g, h);
```

```
> c;
Finitely presented group C on 17 generators
Index in group p4g is 32 = 2^5
Generators as words in group p4g
      C.1 = s^4
      C.2 = r * s^4 * r
      C.3 = (s * r)^4
      C.4 = (s * r * s^-1 * r)^2
      C.5 = (s^-1 * r * s * r)^2
      C.6 = (s^-1 * r)^4
      C.7 = s * r * s^2 * r * s^-1 * r * s^-2 * r
      C.8 = s * r * s^4 * r * s^-1
      C.9 = s^2 * r * s * r * s^-2 * r * s^-1 * r
      C.10 = s^2 * r * s^-1 * r * s^-2 * r * s * r
      C.11 = s^-1 * r * s^2 * r * s * r * s^-2 * r
      C.12 = s^-1 * r * s^4 * r * s
      C.13 = r * s * r * s^4 * r * s^-1 * r
      C.14 = r * s^-1 * r * s^4 * r * s * r
      C.15 = s^2 * r * s^2 * r * s^-2 * r * s^-2 * r
      C.16 = s^2 * r * s^4 * r * s^-2
      C.17 = r * s^2 * r * s^4 * r * s^-2 * r
```

10.4 Quotient Methods

- Abelian quotient
- p-quotient
- Nilpotent quotient
- Soluble quotient

10.4.1 Constructing a Burnside Group

We use the p-quotient algorithm to construct the largest finite 2-generator group having exponent 5.

```
> F := FreeGroup(2);
> q := pQuotient (F, 5, 14: Print := 1, Exponent := 5);
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 5^5
Group: F to lower exponent-5 central class 4 has order 5^8
Group: F to lower exponent-5 central class 5 has order 5^10
Group: F to lower exponent-5 central class 6 has order 5^14
Group: F to lower exponent-5 central class 7 has order 5^18
Group: F to lower exponent-5 central class 8 has order 5^22
Group: F to lower exponent-5 central class 9 has order 5^28
Group: F to lower exponent-5 central class 10 has order 5^31
Group: F to lower exponent-5 central class 11 has order 5^33
Group: F to lower exponent-5 central class 12 has order 5^34
Group completed. Lower exponent-5 central class = 12, order = 5^34
```

10.5 Construction and Presentation of Subgroups

- Low index subgroups (Sims backtrack algorithm)
- Subgroup presentations (Reidemeister-Schreier rewriting)
- Tietze transformations
- Presentation simplification (new Havas-Lian algorithm)

10.5.1 Proving a Deficiency Zero Group to be Infinite

This example shows how the low index subgroup machinery may be used to prove that a group is infinite. The original solution is due to Mike Slattery.

```
> F<x, z> := FreeGroup(2);
> G<x, z> := quo< F | z^3*x*z^3*x^-1, z^5*x^2*z^2*x^2 >;
> G;

Finitely presented group G on 2 generators
Relations
       z^3 * x * z^3 * x^-1 = Id(G)
       z^5 * x^2 * z^2 * x^2 = Id(G)

> LP := LowIndexProcess(G, 30);
> i := 0;
> found := false;
>  while i le 100 and not IsEmpty(LP) do
>       H := ExtractGroup(LP);
>       NextSubgroup($\sim$LP);
>       if not IsFinite(AbelianQuotient(H)) then
>               print "The group G has subgroup:\n", H;
>               print "\nwhose abelian quotient has structure", AQInvariants(H);
>               print "Hence G is infinite.";
>               found := true;
>               break;
>       end if;
>       i +:= 1;
> end while;
> if not found then print "Test fails."; end if;

The group G has subgroup:-
Finitely presented group H on 4 generators
Index in group G is 4 = 2^2
Generators as words in group G
       H.1 = x
       H.2 = z * x * z
       H.3 = z^3
       H.4 = z * x^-1 * z * x * z^-1
whose abelian quotient has structure [ 2, 6, 0 ]
Hence G is infinite.
```

10.5.2 Proving that the Fibonacci Group $F(9)$ is Infinite

The finiteness of the last of the Fibonacci groups, $F(9)$, was settled in 1988 by M.F. Newman using the following result:

Theorem. *Let G be a group with a finite presentation on b generators and r relations, and suppose p is an odd prime. Let d denote the rank of the elementary abelian group $G_1 = [G, G]G^p$ and let e denote the rank of $G_2 = [G_1, G]G^p$. If*

$$r - b < d^2/2 - d/2 - d - e$$

or

$$r - b \leq d^2/2 - d/2 - d - e + (e + d/2 - d^2/4)d/2,$$

then G has arbitrary large quotients of p-power order.

We present a proof that $F(9)$ is infinite using this result. The proof involves constructing presentations for subgroups (function `Rewrite`) and p-quotient calculations (function `pQuotient`).

```
> Left := func< b, r | r - b >;
> Right := func< d, e | d^2 div 2 - d div 2 - d - e +
>   (e + d div 2 - d^2 div 4)*(d div 2) >;
>
> F< x1,x2,x3,x4,x5,x6,x7,x8,x9 > :=
>   Group< x1, x2, x3, x4, x5, x6, x7, x8, x9 |
>   x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6, x5*x6=x7,
>   x6*x7=x8, x7*x8=x9, x8*x9=x1, x9*x1=x2 >;
>
> F;
Finitely presented group F on 9 generators
Relations
    x1 * x2 = x3
    x2 * x3 = x4
    x3 * x4 = x5
    x4 * x5 = x6
    x5 * x6 = x7
    x6 * x7 = x8
    x7 * x8 = x9
    x8 * x9 = x1
    x9 * x1 = x2
> AbelianQuotientInvariants(F);
[ 2, 38 ]
```

Thus the nilpotent quotient of F is divisible by 2 and 19. We examine the 2- and 19-quotients of F.

```
> Q1 := pQuotient(F, 2, 0: Print := 1 );
Class bound of 63 taken
Lower exponent-2 central series for F
Group: F to lower exponent-2 central class 1 has order 2^2
Group: F to lower exponent-2 central class 2 has order 2^3
Group completed. Lower exponent-2 central class = 2, order = 2^3
> Q2 := pQuotient(F, 19, 0: Print := 1 );
```

```
Class bound of 63 taken
Lower exponent-19 central series for F
Group: F to lower exponent-19 central class 1 has order 19^1
Group completed. Lower exponent-19 central class = 1, order = 19^1
```

Thus the nilpotent residual of F has index 152. We try to locate this subgroup. We first take a 2-generator presentation for F.

```
> G := Simplify(F);
> G;
Finitely presented group G on 2 generators
Generators as words in group F
  G.1 = x1
  G.2 = x2
Relations
  G.2 * G.1 * G.2 * G.1 * G.2^2 * G.1 * G.2^2 * G.1^-1 * G.2 * G.1^-2 * G.2 *
  G.1^-2 = Id(G)
  G.1 * G.2^2 * G.1 * G.2 * G.1^2 * G.2^-1 * G.1^2 * G.2^-1 * G.1 * G.2^-1 *
  G.1^2 * G.2^-1 * G.1 * G.2^-1 = Id(G)
> H := ncl< G | (G.1, G.2) >;
> H;
Finitely presented group H
Index in group G is 76 = 2^2 * 19
Subgroup of group G defined by coset table
```

We don't have the full nilpotent residual yet so we try again.

```
> H := ncl< G | (G.1*G.1, G.2) >;
> H;
Finitely presented group H
Index in group G is 152 = 2^3 * 19
Subgroup of group G defined by coset table
```

We have it now.

```
> AbelianQuotientInvariants(H);
[ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
```

The nilpotent H residual has a 5-quotient. We construct a presentation for H and then calculate d and e for its 5-quotient.

```
> K := Rewrite(G, H: Simplify := false);
> KP := pQuotientProcess(K, 5, 1);
> d := FactoredOrder(ExtractGroup(KP))[1][2];
> NextClass(~KP);
> e := FactoredOrder(ExtractGroup(KP))[1][2] - d;
> "d = ", d, "e = ", e;
d =  18 e =  81
> "Right hand side = ", Right(d, e);
Right hand side =  135
```

```
> "Left hand side = ", Left(Ngens(K), #Relations(K));
Left hand side =  151
```

Since Left is greater than Right, this presentation for H doesn't work so we start eliminating generators.

```
> K := Simplify(K: Iterations := 20);
> "Left hand side = ", Left(Ngens(K), #Relations(K));
Left hand side =  89
```

Got it! By Newman's theorem, H has an infinite 5-quotient and so F must be infinite.

Chapter 11

Finite Soluble Groups

11.1 Introduction

For computational purposes it is most effective to represent a soluble group by means of a polycyclic presentation.

11.2 Construction and Characteristic Subgroups

- Representation of a soluble group in terms of a *special AG presentation*
- Construction of a polycyclic presentation for a soluble group given as a permutation group or a matrix group
- Construction of polycyclic presentation for a p-quotient of an fp-group (O'Brien's p-quotient program)
- Split and non-split extensions, wreath products
- Construction of subgroups, quotient groups
- Normalizer, centralizer, normal closure, core of a subgroup
- Testing elements and subgroups for conjugacy
- Intersection of subgroups, Sylow p-subgroup, centre, derived subgroup
- Series: chief, composition, derived, upper central, lower central, elementary abelian, p-central, Jennings
- Hall π-subgroups, Sylow basis, complement basis
- System normalizer, relative system normalizer, Fitting subgroup, Carter subgroup
- System of double coset representatives for a pair of subgroups

11.2.1 A Conjecture of Hawkes and Cossey

We verify an easy example from a paper of Hawkes and Cossey. 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 paper gives an example of a group having derived length 3 in which this result is fal se. Mike Slattery wrote the following Magma code to verify their example:

```
> E := DihedralGroup(GrpPC,5);
> A := CyclicGroup(GrpPC,8);
```

First, we define an action of E on A:

```
> f1 := hom< A -> A | A.1 -> (A.1)^-1 >;
> f2 := hom< A -> A | A.1 -> A.1 >;
```

and create the split extension:

```
> H := Extension(A, E, [f1,f2]);
```

Then we construct a certain H-module:

```
> QH := SylowSubgroup(H,2);
> m1 := TrivialModule(QH,FiniteField(5));
> m2 := Induction(m1, H);
```

and form the split extension of H acting on that module:

```
> G := Extension(m2,H);
> 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
> DerivedLength(G);
3
```

Finally, we check the relevant class sizes:

```
> P := SylowSubgroup(G,5);
> Q := SylowSubgroup(G,2);
> time Maximum({x[2] : x in Classes(G)});
1250
Time: 30.019
> time Maximum({x[2] : x in Classes(P)});
625
Time: 0.090
> time Maximum({x[2] : x in Classes(Q)});
4
Time: 0.009
```

11.2.2 Analysis of a Small Soluble Permutation Group

The following example concerns a soluble permutation group H of order 48. An abstract description of its structure was found by John Cannon and Mike Newman in response to a question raised by Brendan McKay. While it does not yet seem possible to automate such analysis completely, this example shows how mathematical insight supported by calculation may led to a satisfactory description of a group in a few minutes.

```
> H := PermutationGroup< 24 |
>  (2,3)(5,6)(7,8)(9,10)(11,12)(14,15)(16,17)(18,19)(20,21)(23,24),
>  (1,2,7,18,8,9)(3,22,10,20,19,21)(4,6,23,17,24,14)(5,13,15,11,16,12),
>  (1,4)(2,14)(3,15)(5,10)(6,9)(7,24)(8,23)(11,21)(12,20)(13,22)(16,19)(17,18)>;
> Order(H);
48
```

Since the order is 48, the group H is soluble/polycyclic, and it may be helpful to have a polycyclic presentation G:

```
> G := PCGroup(H);
> G;
GrpPC : G of order 48 = 2^4 * 3
PC-Relations:
G.1^2 = Id(G),
G.2^2 = Id(G),
G.3^2 = Id(G),
G.4^2 = Id(G),
G.5^3 = Id(G),
G.3^G.2 = G.3 * G.4,
G.5^G.1 = G.5^2,
G.5^G.3 = G.5^2
```

One can read off: the commutator subgroup is $\langle G.4, G.5\rangle$, and it is abelian and has order 6 – thus cyclic; the commutator quotient is elementary abelian of order 8. Also $\langle G.1, G.2, G.3, G.4\rangle$ is a Sylow 2-subgroup; $\langle G.2, G.3, G.4\rangle$ is dihedral of order 8. Furthermore, $\langle G.5\rangle$ is the Sylow 3-subgroup, and its centralizer has order 24. We find this centralizer C.

```
> C := Centralizer(G, G.5);
```

```
> C;
GrpPC : C of order 24 = 2^3 * 3
PC-Relations:
C.1^2 = Id(C),
C.2^2 = Id(C),
C.3^3 = Id(C),
C.4^2 = Id(C),
C.2^C.1 = C.2 * C.4
```

Thus C has a direct decomposition into $\langle C.3\rangle$ which is cyclic of order 3 and $B = \langle C.1, C.2, C.4\rangle$ which is dihedral of order 8. Hence both these subgroups are characteristic in C and normal in G. Furthermore, B has $\langle C.3, G.5\rangle$ as a complement which is normal and non-abelian of order 6. So G has a direct decomposition into a dihedral subgroup of order 8 and a non-abelian subgroup of order 6.

We find B, and get all the complements of B in G. The function `Complements(G, B)` actually returns conjugacy class representatives for these complements. Finally, we test to see which complements are normal in G.

```
> B := sub<G | C.1, C.2, C.4>;
> B;
GrpPC : B of order 8 = 2^3
PC-Relations:
    B.2^B.1 = B.2 * B.3
> Com := Complements(G, B);
> [IsNormal(G, X) : X in Com];
[ false, true, true, false ]
```

Thus there are 4 complements, of which two are normal.

11.3 Subgroup Structure, Automorphisms and Representations

- Maximal subgroups, Frattini subgroup
- Conjugacy classes of elements
- Conjugacy classes of complements of a normal subgroup
- Normal subgroups
- Conjugacy classes of subgroups, poset of subgroup classes
- Automorphism group and isomorphism of p-groups (explicit isomorphism returned)
- p-group generation (Eamonn O'Brien's program)
- Character table
- Irreducible modular representations
- Databases: 2-groups of order dividing 256; Non-abelian soluble groups of order less than 100

11.3.1 Maximal subgroups of B(2,6)

We load a definition of the Burnside group $B(2, 6)$ from a Magma database `solgps` containing definitions of various interesting soluble groups. $B(2, 6)$ is a two-generator group, the unique group of order $2^{28} \cdot 3^{25}$ and exponent 6.

```
> load solgps;
Loading "/home/magma/v4/LIBS/solgps/solgps"
For a summary of the contents, type
        ?solgps

> G := B26();

The following is a pc-presentation of the two generator Burnside group
B(2,6) of exponent 6 and order 2^28*3^25.

A group with factorized order [ <2, 28>, <3, 25> ]
has been constructed
```

We look at the orders of the terms of the derived series for G:

```
> ds := DerivedSeries(G);
> [ FactoredOrder(x) : x in ds ];
[
    [ <2, 28>, <3, 25> ],
    [ <2, 26>, <3, 23> ],
    [ <2, 18>, <3, 13> ],
    []
]
```

What does a chief series for G look like?

```
> cs := ChiefSeries(G);
> [ FactoredOrder(x) : x in cs ];
[
    [ <2, 28>, <3, 25> ],
    [ <2, 27>, <3, 25> ],
    [ <2, 26>, <3, 25> ],
    [ <2, 26>, <3, 24> ],
    [ <2, 26>, <3, 23> ],
    [ <2, 26>, <3, 22> ],
    [ <2, 26>, <3, 21> ],
    [ <2, 26>, <3, 20> ],
    [ <2, 26>, <3, 19> ],
    [ <2, 26>, <3, 18> ],
    [ <2, 26>, <3, 17> ],
    [ <2, 26>, <3, 16> ],
    [ <2, 26>, <3, 15> ],
    [ <2, 26>, <3, 14> ],
    [ <2, 26>, <3, 13> ],
    [ <2, 26>, <3, 12> ],
    [ <2, 26>, <3, 11> ],
    [ <2, 26>, <3, 10> ],
    [ <2, 26>, <3, 9> ],
    [ <2, 26>, <3, 8> ],
    [ <2, 26>, <3, 7> ],
    [ <2, 26>, <3, 6> ],
    [ <2, 26>, <3, 5> ],
```

```
    [ <2, 26>, <3, 4> ],
    [ <2, 26>, <3, 3> ],
    [ <2, 26>, <3, 2> ],
    [ <2, 26>, <3, 1> ],
    [ <2, 26> ],
    [ <2, 20> ],
    [ <2, 14> ],
    [ <2, 8> ],
    [ <2, 6> ],
    [ <2, 4> ],
    [ <2, 2> ],
    []
]
```

Finally, we determine the maximal subgroups of G and print their indices in G.

```
> Maxs := MaximalSubgroups(G);
> #Maxs;
35
> Sort([ Index(G, H) : H in Maxs] );
[ 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 64, 64, 64, 64, 64, 64,
 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64 ]
```

So we see there are 35 proper maximal subgroups of G and that the smallest has index 64.

11.3.2 Structural Analysis of a Polycyclic Group

We analyze the structure of a polycyclic group. The group arose in the context of an investigation of groups of deficiency zero (by Havas, Newman and O'Brien). The group is visibly an extension of a 2-group by a 3-group. The 3-group is known, from earlier work, to be a group of order 3^5 and class 3.

The analysis below is a sanitised version of an analysis first done by Alice Niemeyer and Mike Newman. Some polishing is due to Allan Steel.

```
> G<a,b,c,d,e,f,g,h,i,j,k,l,m,n> :=
> PolycyclicGroup < a,b,c,d,e,f,g,h,i,j,k,l,m,n |
> a^3 = d, b^3 = d^2*e, c^3, d^3, e^3,
> f^2 = l, g^2 = l, h^2 = l*m, i^2 = l*m, j^2 = l*n, k^2 = l*n, l^2, m^2, n^2,
> b^a=b*c, c^a=c*d*e, d^a=d, e^a=e, f^a=j*k, g^a=j, h^a=g, i^a=f*g,
> j^a=i, k^a=h*i, l^a=l*n, m^a=n, n^a=m*n,
> c^b=c*d, d^b=d, e^b=e, f^b=j*k, g^b=j, h^b=f, i^b=g, j^b=h,
> k^b=i, l^b=l*n, m^b=n, n^b=m*n,
> d^c=d, e^c=e, f^c=f, g^c=g, h^c=i, i^c=h*i, j^c=j*k, k^c=j, l^c=l, m^c=m,
> n^c=n, e^d=e, f^d=g, g^d=f*g, h^d=i, i^d=h*i, j^d=k, k^d=j*k, l^d=l,
> m^d=m, n^d=n, f^e=f, g^e=g, h^e=h, i^e=i, j^e=j, k^e=k, l^e=l, m^e=m,
> n^e=n, g^f=g*l, h^f=h, i^f=i, j^f=j, k^f=k, l^f=l, m^f=m, n^f=n,
> h^g=h, i^g=i, j^g=j, k^g=k, l^g=l, m^g=m, n^g=n,
> i^h=i*l*m, j^h=j, k^h=k, l^h=l, m^h=m, n^h=n,
> j^i=j, k^i=k, l^i=l, m^i=m, n^i=n,
```

```
> k^j=k*l*n, l^j=l, m^j=m, n^j=n,
> l^k=l, m^k=m, n^k=n,
> m^l=m, n^l=n,
> n^m=n >;
>
>
> S2 := SylowSubgroup(G, 2);
```

One can read off that the Sylow 2-subgroup has lower 2-central class 2. The next few lines confirm this.

```
> D := CommutatorSubgroup(S2, S2);
> IsElementaryAbelian(S2/D);
true
> IsElementaryAbelian(D);
true
>
> S3 := SylowSubgroup(G, 3);
```

The action of the Sylow 3-subgroup on S2/D is a 3-subgroup of GL$(6, 2)$ and so a subgroup of the wreath product $C_3 \wr C_3$.

```
> Order(Centre(G));
6
```

The 3-part of the centre has order 3, so the faithful action on S2/D is $C_3 \wr C_3$ and S2/D must be irreducible. The next step confirms that.

```
> IsIrreducible(GModule(G, S2, D));
true
```

This module will reduce under the action of at least one of the maximal subgroups. Finding this reduction helps the analysis. Since the Sylow 3-subgroup can be generated by 2 elements it has 4 maximal subgroups.

```
> M3 := MaximalSubgroups(S3);
```

We test what happens. Note that for each group A in M3 we need to create $H = \langle S2, A\rangle$. It is not possible to use `GModule(A, S2, D)`, since the second argument would not be a subgroup of the first one.

```
> for i in [1..4] do
>     A := M3[i];
>     H := sub<G | S2, A>;
>     Mod, phi := GModule(H, S2, D);
>     if not IsIrreducible(Mod) then
>         print "subgroup", i, "acts reducibly";
>         break i;
```

```
>     else
>         print "subgroup", i, "acts irreducibly";
>     end if;
> end for;
subgroup 1 acts irreducibly
subgroup 2 acts irreducibly
subgroup 3 acts reducibly
```

We have found a subgroup A (the third maximal subgroup) that acts reducibly.

Next, we look for 'small' normal subgroups of H in S2. The idea is to take a non-zero element of a minimal H-submodule of S2/D, pull it back into S2 and calculate the A-admissible subgroup T1 generated by it. Unfortunately not all the A-admissibles have the same order. In this case one gets orders 8 and 16. So we take the first of order 8. Because there is randomness in one of the routines involved in `MinimalSubmodules`, it is not predictable when this will occur. This explains why the output may vary from run to run.

```
> Mins := MinimalSubmodules(Mod);
>
> for i in [1..3] do
>     tt := G ! (Mins[i].1 @@ phi);
>     print tt;
>     T1 := SylowSubgroup(sub<G | tt, A>, 2);
>     print T1;
>     if Order(T1) eq 8 then
>         break i;
>     end if;
> end for;
j
GrpPC : T1 of order 8 = 2^3
PC-Relations:
T1.1^2 = T1.3,
T1.2^2 = T1.3,
T1.2^T1.1 = T1.2 * T1.3
```

T1 is visibly quaternion (of order 8). Find an element of S3\A.

```
> [ G ! A.i : i in [1..4]];
[ a * b^2, c, d * e, e ]
```

Such an element is a. Calculate the conjugates of T1 under a.

```
> T2 := T1 ^ a;
> T3 := T2 ^ a;
> Order(sub<G | T1, T2, T3>);
512
```

The conjugates of T1 under a span S2 . So S2 is the direct sum of three copies of the quaternions. Now explore the action of A on the Ti more closely. First compute a centraliser.

```
> C23 := Centraliser(A, sub<G | T2, T3>);
```

```
> C23;
GrpPC : C23 of order 9 = 3^2
PC-Relations:
C23.1^3 = C23.2
>
> Order(Centraliser(C23, T1));
3
```

Thus this centraliser is cyclic of order 9. Together with T1 it generates U1 which is a split extension of the quaternions by a cyclic group of order 9 acting as an automorphism of order 3. So the group is an extension of a central subgroup of order 3 by $\mathrm{SL}(2,3)$.

```
> U1 := sub<G | T1, C23>;
> U2 := U1^a;
> U3 := U2^a;
> Order(sub<G | U1, U2, U3>);
41472
```

Thus the conjugates of U1 span H (which has index 3 in G). In fact H is a central product of these conjugates.

```
> Y := SylowSubgroup(Centre(H), 3);
> Y;
GrpPC : Y of order 3
PC-Relations:
Y.1^3 = Id(Y)
>
> V := U1 meet U2;
> V;
GrpPC : V of order 3
PC-Relations:
V.1^3 = Id(V)
>
> V eq Y;
true
```

Thus G can be viewed as built from conjugates of U1 under a with a^3 in H but not the centre of H.

Chapter 12

Permutation Groups

12.1 Contruction and Actions

Magma contains a large amount of machinery for dealing with permutation groups of finite degree. The following list summarises various constructions of these groups available in Magma.

- Permutation representations for classical groups, e.g. $\mathrm{PGL}(n,q)$, $\mathrm{PSp}(n,q)$, $\mathrm{PSU}(n,q^2)$, $P\Omega(n,q)$
- Construction of wreath products with both types of action
- Orbits on points, sets of points, and sequences of points
- Systems of imprimitivity
- Induced actions on G-sets
- Base and strong generating set: Sims-Schreier algorithm, random Schreier algorithm, Todd-Coxeter Schreier algorithm, Brownie-Cannon-Sims verification
- Stabilizer of a point, set of points, sequence of points, partition
- Automorphism groups of graphs, codes and designs
- Homomorphisms induced by actions on orbits and systems of imprimitivity
- Random element generation: Holt-Leedham-Green-O'Brien algorithm
- Databases: Transitive groups up to degree 15 (Butler), primitive groups up to degree 50 (Sims), irreducible soluble subgroups of $\mathrm{GL}(n,p)$ for $p^n < 256$ (Short), simple groups up to order a million (Campbell and Robertson), sporadic simple groups

12.1.1 Shuffle Groups

Consider a pack of $3n$ cards, which is shuffled as follows: The pack is divided into three piles each containing n cards and the piles are placed in a row in the order they are removed from the deck. Now the three piles are permuted among themselves, and finally the deck is reassembled by taking the first card from the first pile, the second card from the second pile, and so on. We wish to know if such a shuffle is *fair* in the sense that any card may appear in any position after a finite number of such shuffles.

For further details, see Steve Medvedoff, Kent Morrison, Groups of Perfect Shuffles, *Mathematics Magazine* **60 no. 1** (Feb 1987), 3–14, and also a paper cited by it: P. Diaconis, R.L. Graham, W.M. Kantor, The mathematics of perfect shuffles, *Adv. Appl. Math.* **4** (1983), 175–196.

We give a function `ShuffleGroup` which produces generators for the group of permutations of the cards defined by this shuffle, and returns this group.

```
ShuffleGroup := function(n)

    m := 3 * n;
    G := SymmetricGroup(m);

    /* If the three piles are labelled a, b and c, we use p = (a, c) and
    q = (a, b, c) to generate the permutations of the piles */
    p := &*[G | (i, i + 2 * n) : i in [1 .. n]];
    q := &*[G | (i, i + n, i + 2 * n) : i in [1 .. n]];

    /* Form the permutation s that corresponds to the interleaving step */
    s := [((i-1) mod 3 ) * n + (i-1) div 3 + 1 : i in [1 .. m ]];

    return sub<G | p, q, s>;

end function;
```

We now determine the degrees of those shuffle groups which do not correspond to perfect shuffles, for n in the range 1 to 120.

```
> not_perfect := [];
> for i := 1 to 40 do
>     G := ShuffleGroup(i);
>     if not IsSymmetric(G) then
>         not_perfect cat:= [ Degree(G) ];
>     end if;
> end for;
> not_perfect;
[ 9, 12, 24, 27, 36, 48, 60, 72, 81, 84, 96, 108, 120 ]
```

12.1.2 Construction of the Design Associated with M_{24}

Starting with generators for the Mathieu group M_{24}, we construct the associated 5-$(24, 5, 1)$ block design.

```
> 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)>;
>
> 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, 7, 9, 10, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24 },
    GSet{ 8, 11, 13 }
]
```

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};
> ds := SetToSequence(B ^ M24);
> D := Design<5, 24 | ds >;
> D;
5-(24, 8, 1) Design with 759 blocks
```

Finally, we compute the stabilizer of the block B.

```
> SB :=Stabilizer(M24, B);
> SB;
Permutation group SB acting on a set of cardinality 24
Order = 322560 = 2^10 * 3^2 * 5 * 7
    (1, 2)(7, 22)(9, 16)(10, 19)(11, 13)(12, 21)(14, 24)(15, 18)
    (2, 3)(7, 24)(9, 14)(11, 13)(15, 23)(16, 22)(17, 21)(19, 20)
    (3, 4)(7, 19)(10, 22)(11, 13)(12, 14)(15, 18)(17, 20)(21, 24)
    (4, 5)(7, 22)(9, 14)(10, 18)(11, 13)(15, 19)(16, 24)(20, 23)
    (5, 8)(7, 24)(9, 16)(10, 12)(11, 13)(14, 22)(17, 20)(19, 21)
    (7, 17, 22)(8, 11, 13)(9, 14, 12)(10, 20, 19)(15, 23, 18)(16, 21, 24)
    (6, 12)(7, 23)(9, 14)(10, 18)(15, 24)(16, 19)(17, 21)(20, 22)
    (6, 17)(7, 22)(9, 18)(10, 14)(12, 21)(15, 16)(19, 24)(20, 23)
    (6, 14)(7, 16)(9, 12)(10, 17)(15, 22)(18, 21)(19, 23)(20, 24)
    (6, 15)(7, 10)(9, 20)(12, 24)(14, 22)(16, 17)(18, 23)(19, 21)
```

The function `CompositionFactors` gives us the structure of this stabilizer.

```
> CompositionFactors(SB);
    G
    |  Alternating(8)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
```

12.2 Subgroup Structure of Permutation Groups

The key concept for representing a permutation group is that of a base and strong generating set (BSGS). 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 several million. The ability to construct a BSGS, coupled with the use of algorithms that make heavy use of the classification theorem for finite simple groups, opened up the possibility of analyzing the structure (e.g. determining the composition factors) for short base groups of degree up to a million.

- O'Nan-Scott decomposition of a primitive group
- Abelian quotient, soluble quotient
- Composition series, composition factors, chief series, chief factors
- Characteristic series: derived, upper central, lower central, elementary abelian, p-central, Jennings
- Characteristic subgroups: Centre, derived subgroup, Fitting subgroup, $O_p(G)$, radical, socle, solvable residual
- Subgroup constructions: Centralizer, normalizer, intersection, normal closure, core
- Sylow p-subgroup (by reduction to a simple group)
- Testing elements and subgroups for conjugacy
- Normal subgroups
- Conjugacy classes of elements
- Conjugacy classes of subgroups, poset of subgroup classes
- Presentation on given generators (for groups of moderate order)
- Character table
- Irreducible representations (for groups of moderate order)
- KG-module corresponding to an elementary abelian section
- First and second cohomology groups, split and non-split extensions of a group by a module (D. Holt's package)

12.2.1 Chief Series of Rubik's $4 \times 4 \times 4$ cube

We illustrate the `ChiefSeries` function by applying it to the group of transformations of the $4 \times 4 \times 4$ Rubik cube. This is represented as a permutation group on 72 letters. The `ChiefSeries` function returns two results, the actual chief series of normal subgroups of G, and a second series identifying the chief factors.

```
> load rubik444;
Loading "/home/magma/v4/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
> cs,cf:=ChiefSeries(G);
> cf;
    G
    |  Cyclic(2)
    *
```

```
    |  Alternating(24)
    *
    |  Cyclic(2)
    *
    |  Alternating(24)
    *
    |  Alternating(8)
    *
    |  Cyclic(3) (7 copies)
    1
```

We find that the chief factors of the group are as shown above. The fifth term of the chief series itself is the normal subgroup of G isomorphic to an elementary abelian group of order 3^7 extended by A_8.

```
> cs[5];
Permutation group acting on a set of cardinality 72
Order = 44089920 = 2^6 * 3^9 * 5 * 7
  (67, 69, 68)(70, 71, 72)
  (64, 67, 70)(65, 69, 72)(66, 68, 71)
  (61, 64, 70)(62, 66, 71)(63, 65, 72)
  (58, 61, 72)(59, 63, 71)(60, 62, 70)
  (55, 58, 70)(56, 59, 72)(57, 60, 71)
  (52, 72, 59, 56, 54, 71, 60, 57, 53, 70, 58, 55)(64, 69, 66, 67, 65, 68)
  (49, 60, 57, 53, 51, 59, 56, 54, 50, 58, 55, 52)(61, 68, 63, 67, 62, 69)
```

This group represents the possible tranformations on the corners of the cube. In terms of positioning the corner cubes every even permutation of them is possible. Each corner cube can be oriented three ways. The orientation of seven of the corner cubes may be chosen freely, and this will determine the orientation of the eighth corner.

12.2.2 Subgroup lattice

We create the subgroup lattice of $\mathrm{A\Gamma L}(1, 8)$ 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 ]
```

12.3 Representations and Cohomology

- Character table
- Irreducible representations (for groups of moderate order)
- KG-module corresponding to an elementary abelian section
- First and second cohomology groups, split and non-split extensions of a group by a module (D. Holt's package)

Chapter 13

Matrix Groups

13.1 Constructions of Matrix Groups

Below are listed some of the constructions and elementary operations on matrix groups available in Magma.

- Generators for linear groups: $\mathrm{GL}(n, q)$, $\mathrm{SL}(n, q)$
- Generators for symplectic groups: $\mathrm{Sp}(n, q)$
- Generators for unitary groups: $\mathrm{GU}(n, q)$, $\mathrm{SU}(n, q)$
- Generators for orthogonal groups: $\mathrm{GO}(2n+1, q)$, $\mathrm{SO}(2n+1, q)$, $\Omega(2n+1, q)$, $\mathrm{GO}^+(2n, q)$, $\mathrm{SO}^+(2n, q)$, $\Omega^+(2n, q)$, $\mathrm{GO}^-(2n, q)$, $\mathrm{SO}^-(2n, q)$, $\Omega^-(2n, q)$
- Generators for all exceptional families of groups of Lie type except $E(8)$.
- Direct product, tensor wreath product, tensor power
- Construction of semi-linear groups
- Random element generation (Cellar, Leedham-Green, Murray, O'Brien algorithm)
- Order of a matrix (Leedham-Green algorithm for finite fields)
- Test whether a matrix has infinite order
- Tests for irreducibility, absolute irreducibility, semi-linearity
- Decomposition of a subgroup of $\mathrm{GL}(n, q)$ into primitive components
- Test whether a group has infinite order
- Base and strong generating set: Random Schreier algorithm, Todd-Coxeter Schreier algorithm, Murray-O'Brien base selection strategy
- Orbit, stabilizer of a vector or subspace

13.1.1 Random generation of matrix group elements

We use the `RandomProcess` 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..10] ];
[
    [ <3, 1>, <5, 2>, <11, 1>, <17, 1>, <31, 1>, <41, 1>, <257, 1>, <61681, 1>
    ],
```

```
    [ <3, 2>, <5, 1>, <7, 1>, <13, 1>, <17, 1>, <37, 1>, <73, 1>, <109, 1>,
    <241, 1>, <433, 1>, <38737, 1> ],
    [ <3, 1>, <5, 2>, <11, 1>, <17, 1>, <31, 1>, <41, 1>, <257, 1> ],
    [ <3, 2>, <5, 2>, <7, 1>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>, <257,
    1>, <65537, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <13, 1>, <17, 1>, <19, 1>, <37, 1>, <73, 1>, <109,
    1>, <241, 1>, <433, 1>, <38737, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <13, 1>, <17, 1>, <29, 1>, <43, 1>, <113, 1>,
    <127, 1>, <241, 1> ],
    [ <3, 1>, <5, 1>, <7, 1>, <13, 1>, <17, 1>, <257, 1>, <65537, 1> ],
    [ <5, 1>, <11, 1>, <17, 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, 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> ]
]
```

13.2 Structure of a Matrix Group

The matrix group facilities are mainly restricted to finite groups since there are, as yet, few algorithms of general interest known for infinite groups. Techniques for working with finite matrix groups divide into methods for groups of small degree and methods for groups of large degree. Our initial concern is the development of a module for dealing with finite matrix groups of small degree. In practice, this means degrees up to 30 or so, over small rings. However, this case is important, since algorithms for dealing with larger degree groups defined over finite fields will ultimately reduce down to this case. The small degree technique involves looking for some sequences of objects (subspaces and vectors) in the underlying vector space that defines a stabilizer chain which has the property that the basic orbits are not excessively large. Thus, we have a concept of a base and strong generating set (BSGS) similar to that employed in the case of permutation groups. Once such a BSGS is available, analogues of the permutation group backtrack searches for centralizer, normalizer etc may be described.

- Homomorphism induced by action of a reducible group on G-invariant submodule and its quotient module
- Homomorphisms induced by actions on orbits of vectors and subspaces
- Homomorphism induced by action on the cosets of a subgroup
- Construction of subgroups, quotient groups
- Abelian quotient, soluble quotient
- Composition series, composition factors, chief series, chief factors
- Characteristic series: Derived, upper central, lower central, elementary abelian, p-central, Jennings
- Characteristic subgroups: Centre, derived subgroup, Fitting subgroup, $O_p(G)$, radical, solvable residual
- Subgroup constructions: Centralizer, intersection, normal closure, core
- Sylow p-subgroup
- Testing elements for conjugacy
- Conjugacy classes of elements
- Presentation on given generators (for groups of moderate order)

- Character table
- Molien series
- Ring of invariants

13.2.1 Bravais Subgroups

In this example we compute the Bravais subgroups of the finite integral matrix group $W(E_6)$.

```
> L := CoordinateLattice(Lattice("E", 6)); // W(E_6)
> L;
Standard Lattice of rank 6 and degree 6
Inner Product Matrix:
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2 -1  0 -1]
[ 0  0 -1  2 -1  0]
[ 0  0  0 -1  2  0]
[ 0  0 -1  0  0  2]
> SV := ShortestVectors(L);
> G := AutomorphismGroup(L);
> G;
MatrixGroup(6, Integer Ring) of order 103680 = 2^8 * 3^4 * 5
Generators:
    [ 1  1  1  0  0  0]
    [-1 -2 -3 -2 -1 -1]
    [ 1  1  2  2  1  1]
    [ 0  1  0  0  0  0]
    [ 0  0  1  0  0  1]
    [-1 -1 -1 -1  0 -1]

    [ 1  1  1  1  1  0]
    [ 0  0 -1 -1 -1  0]
    [ 0 -1  0  0  0  0]
    [ 0  1  1  0  0  0]
    [ 0  0  0  1  0  0]
    [-1 -1 -2 -1  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]
    [ 0  0  0  0  1  0]
    [-1 -2 -3 -2 -1 -1]
```

We now find the subgroups of G. Currently, the subgroup function only works for permutation groups. In general the next line does not yield a faithful action it does, though, for irreducible representations

```
> phi, P := OrbitAction(G, SV[1]);
> time Sub := Subgroups(P);
```

```
Time: 127.949
> print "Computed", #Sub, "subgroups";
Computed 1503 subgroups
> Bravais := [];
> t := Cputime();
> for S in Sub do
>       U := S`subgroup @@ phi;
>       A := BravaisGroup(U);
>       if Order(A) eq Order(U) then
>               Append(~Bravais, U);
>       end if;
> end for;
> print "\nFound", #Bravais, "Bravais subgroups";

Found 86 Bravais subgroups

> print [ #H : H in Bravais ];
[ 2, 4, 4, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 8, 8, 12, 12, 12, 12,
12, 12, 20, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 24, 24,
24, 24, 24, 24, 24, 36, 36,40, 32, 32, 32, 32, 32, 32, 32, 32, 48,
48, 48, 48, 48, 72, 72, 72, 64, 64, 64, 96, 96, 96, 96, 144, 144,
144, 240, 128, 192, 192, 192, 288, 432, 480, 384, 384, 864, 768,
2880, 2304, 3840, 103680 ]


> print [ H : H in Bravais | #H eq 480 ];
[
    MatrixGroup(6, Integer Ring) of order 480 = 2^5 * 3 * 5
    Generators:
        [ 1  1  0  0  0  0]
        [ 0  0  1  1  0  1]
        [-1 -1 -2 -2 -1 -1]
        [ 0 -1 -1  0  0 -1]
        [ 0  1  1  1  1  1]
        [ 1  1  2  1  1  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]
        [ 0  0  0  0 -1  0]
        [ 0  0  0  0  0 -1]

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

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

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

13.3 Decomposition of Matrix Groups over Finite Fields

Consider a matrix group G acting on the finite dimensional KG-module V over a finite field K. According to a theorem of Aschbacher (M. Aschbacher, On the maximal subgroups of the finite classical groups, *Invent. Math.* 76 (1984) 469-514.), G falls into one of nine classes. The facilities listed here attempt to decide into which such class a given group falls.

- Determine whether G acts reducibly on V
- Determine whether G acts semilinearly over an extension field of K
- Determine whether G acts imprimitively on V
- Determine whether G preserves a nontrivial tensor-product decomposition of V
- Determine whether G acts (modulo scalars) linearly over a proper subfield of K
- Determine whether G contains the special linear group, or one of the classical groups in its natural action over K

Chapter 14

Coxeter Groups

14.1 Summary of Facilities

Finite Coxeter groups are implemented as a subclass of permutation groups so that they inherit all the operations for permutation groups as well as having many specialized functions. This module was implemented by Don Taylor. Frank Lübeck and the Chevie team provided helpful assistance.

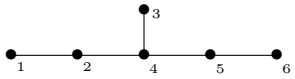
- Cartan matrix corresponding to a given Dynkin diagram
- Construction of a Coxeter group from a root system or Cartan matrix
- Dynkin diagram of a Cartan matrix or Coxeter group
- Root system for a Coxeter group
- Unique long (short) root of greatest height
- Short root of maximal height
- Reflections in Coxeter group
- Reflection subgroup
- Reduced representatives for cosets of the reflection subgroup
- Actions on roots and co-roots
- Coxeter group as a real reflection group
- Killing form of the Cartan algebra associated with a given Weyl group
- Root elements
- Fundamental roots and their negatives of a simple Lie algebra of given type and rank
- Lie algebra of a Chevalley group as a structure constant algebra
- Degree of a representation with specified weight
- The BN-pair for a Chevalley group

14.2 Constructing the split octonions

The purpose of this example is to construct an 8-dimensional module for the Lie algebra of type G_2 and to define a multiplication on it which turns it into an alternative algebra. The algebra will turn out to be the octonions and the elements of G_2 will act as derivations. The example was constructed by Don Taylor (University of Sydney).

14.2.1 The Lie algebra of type D_4

Let W be the Coxeter group of type E_6, let Φ be its root system and let $\Delta = \{\alpha_1, \alpha_2, \ldots, \alpha_6\}$ be the set of simple roots labelled according to the Dynkin Diagram



There are 36 positive roots and 24 of these lie outside the subsystem (of type D_4) spanned by α_2, α_3, α_4 and α_5. These 24 roots are partitioned into 3 subsets of size 8 according to the values of their first and last coordinates. We label these sets X, Y and Z.

```
> W := CoxeterGroup("E",6);
> Phi := RootSystem(W);
> X := Reverse({@ x : x in Phi | x[1] eq 1 and x[6] eq 0 @});
> Y := Reverse({@ x : x in Phi | x[1] eq 0 and x[6] eq 1 @});
> Z := Reverse({@ x : x in Phi | x[1] eq 1 and x[6] eq 1 @});
> #X, #Y, #Z;
8 8 8
```

We have reversed the order of the roots so that, in each case, they are ordered from highest to lowest.

Choose a Chevalley basis $\{ e_\alpha \mid \alpha \in \Phi \} \cup \{ h_\beta \mid \beta \in \Delta \}$ for the Lie algebra of type E_6. The Lie algebra of type D_4 is generated (as a Lie algebra) by the root vectors $e_{\pm\alpha_i}$, $2 \leq i \leq 5$. In MAGMAwe shall use `d4eX[i]` to denote the matrix representing the action of the root vector of the i-th simple root of D_4 acting on the vector space $V_x = \{ e_\alpha \mid \alpha \in X \}$. Similarly, `d4fX` will be the sequence of matrices representing the actions of the root vectors of the negatives of the simple roots.

```
> d4eX := [LieRootMatrix(W,Phi[i],X) : i in [2..5]];
> d4fX := [LieRootMatrix(W,-Phi[i],X) : i in [2..5]];
```

Let V_y and V_z be the vector spaces with bases indexed by the roots in Y and Z respectively and let `d4eY`, `d4fY`, `d4eZ`, `d4fZ` be the corresponding sequences of matrices representing the action of the simple root vectors of D_4.

```
> d4eY := [LieRootMatrix(W,Phi[i],Y) : i in [2..5]];
> d4fY := [LieRootMatrix(W,-Phi[i],Y) : i in [2..5]];
> d4eZ := [LieRootMatrix(W,Phi[i],Z) : i in [2..5]];
> d4fZ := [LieRootMatrix(W,-Phi[i],Z) : i in [2..5]];
```

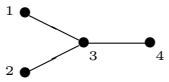
The Cartan subalgebra H of D_4 has basis $\{h_2, h_3, h_4, h_5\}$, where $h_i = h_{\alpha_i}$, and its dual has basis $\{\alpha_2, \alpha_3, \alpha_4, \alpha_5\}$. Moreover, for $h \in H$ and $\alpha \in X$ we have $[h\, e_\alpha] = \alpha(h) e_\alpha$ and so the weights of the module V_x are simply the roots $\alpha \in X$ restricted to H.

The Cartan matrix of W has entries $\langle \alpha_i, \alpha_j \rangle$ and since the form $\langle -, - \rangle$ is linear in the first variable the values of $\langle \alpha, \alpha_j \rangle$ are given by the product of the row vector α by the matrix C.

```
> C := CartanMatrix(W);
```

```
> X[1];
(1 1 2 2 1 0)
> X[1]*C;
( 0 0 1 0 0 -1)
```

This shows that, as a D_4 module, X has highest weight λ_2 according to the following labelling of the D_4 Dynkin diagram, where λ_1, λ_2, λ_3 and λ_4 are the fundamental weights.



We carry out similar calculations with Y and Z.

```
> Y[1];
(0 1 1 2 2 1)
> Y[1]*C;
(-1  0  0  0  1  0)
> Z[1];
(1 2 2 3 2 1)
> Z[1]*C;
(0 1 0 0 0 0)
```

Thus Y has highest weight λ_4 and Z has highest weight λ_1.

14.2.2 The Lie algebra of type G_2

The Lie algebra of type G_2 may be regarded as the subalgebra of D_4 generated by $e_A = e_{\alpha_2} + e_{\alpha_3} + e_{\alpha_5}$, $e_B = e_{\alpha_4}$, $e_{-A} = e_{-\alpha_2} + e_{-\alpha_3} + e_{-\alpha_5}$ and $e_{-B} = e_{-\alpha_4}$.

As D_4-modules the spaces V_x, V_y and V_z are not isomorphic. However they become isomorphic when considered as G_2-modules. We note in passing that D_4 has an automorphism of order 3 which permutes the modules V_x, V_y and V_z. It should be possible to obtain the "principle of local triality" directly from this and then obtain the matrices P and Q (defined below).

The matrices for the G_2 action on V_x are

```
> eX := [d4eX[1]+d4eX[2]+d4eX[4],d4eX[3]];
> fX := [d4fX[1]+d4fX[2]+d4fX[4],d4fX[3]];
```

Similarly the action on V_y and V_z is given by

```
> eY := [d4eY[1]+d4eY[2]+d4eY[4],d4eY[3]];
> fY := [d4fY[1]+d4fY[2]+d4fY[4],d4fY[3]];
>
> eZ := [d4eZ[1]+d4eZ[2]+d4eZ[4],d4eZ[3]];
> fZ := [d4fZ[1]+d4fZ[2]+d4fZ[4],d4fZ[3]];
```

The modules V_x, V_y and V_z are reducible and split into the sum of an irreducible 7-dimensional submodule and a 1-dimensional complement. From highest weight theory the 7-dimensional submodule has a basis consisting of vectors of the form $e_{\beta_1}^{i_1} \cdots e_{\beta_m}^{i_m} v^+$, where v^+ is a maximal vector and $\beta_i \in \{-A, -B\}$. To set this up in MAGMAwe need a vector space of dimension 8.

```
> V := VectorSpace(RationalField(),8);
> B := Basis(V);
```

In each of V_x, V_y and V_z the highest weight vector is the first basis element. Thus to describe a basis for the 7-dimensional submodule it is enough to give a sequence of root elements with the property that the $(i+1)$-st basis element is obtained by applying the i-th root element to the i-th basis element.

We see directly from the forms of `fX[1]` and `fX[2]` that for V_x we may take

```
> ss := [1,2,1,1,2,1];
> Bx := [i eq 0 select B[1] else Self(i)*fX[ss[i]] : i in [0..6]];
> Bx;
[
    V: (1  0  0  0  0  0  0  0),
    V: (0 -1  0  0  0  0  0  0),
    V: (0  0  1  0  0  0  0  0),
    V: (0  0  0 -1  1  0  0  0),
    V: (0  0  0  0  0 -2  0  0),
    V: (0  0  0  0  0  0 -2  0),
    V: (0  0  0  0  0  0  0 -2)
]
```

We apply the same construction to V_y

```
> By := [i eq 0 select B[1] else Self(i)*fY[ss[i]] : i in [0..6]];
> By;
[
    V: (1  0  0  0  0  0  0  0),
    V: (0 -1  0  0  0  0  0  0),
    V: (0  0  1  0  0  0  0  0),
    V: (0  0  0 -1 -1  0  0  0),
    V: (0  0  0  0  0  2  0  0),
    V: (0  0  0  0  0  0 -2  0),
    V: (0  0  0  0  0  0  0  2)
]
```

It turns out that the matrices for V_y and V_z are identical:

```
> eY eq eZ, fY eq fZ;
true true
```

Writing $x_1, \ldots, x_8$ to denote the basis vectors of V_x (similarly for V_y and V_z), the 1-dimensional G_2-submodule of V_x is spanned by $x_4+x_5 = (0,0,0,1,1,0,0,0)$ and that of V_y (resp. V_z) is spanned

by $y_4 - y_5 = (0,0,0,1,-1,0,0,0)$ (resp. $z_4 - z_5$). Thus we may define a G_2-module isomorphism $\varphi : V_z \to V_x$ with matrix

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}(\lambda+1) & \frac{1}{2}(\lambda-1) & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2}(\lambda-1) & -\frac{1}{2}(\lambda+1) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

where $\lambda \neq 0$ can be chosen arbitrarily. Similarly, we define a G_2-module homomorphism $\psi : V_z \to V_y$ with matrix

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}(\mu+1) & -\frac{1}{2}(\mu-1) & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2}(\mu-1) & \frac{1}{2}(\mu+1) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where $\mu \neq 0$.

14.2.3 The octonion algebra

We shall attempt to determine λ and μ so that the product defined on V_z by

$$u \star v = [u^\varphi \, v^\psi]$$

turns V_z into an alternative algebra. The identity element must be a multiple of $z_4 - z_5$ and we have $(z_4 - z_5)^\varphi = \lambda(x_4 + x_5)$ and $(z_4 - z_5)^\psi = \mu(y_4 - y_5)$.

```
> for i := 1 to 8 do
for> c4 := LieStructureConstant(W,X[4],Y[i]);
for> c5 := LieStructureConstant(W,X[5],Y[i]);
for> z4 := c4 eq 0 select 0 else X[4]+Y[i];
for> z5 := c5 eq 0 select 0 else X[5]+Y[i];
for> <i,c4,c5,z4,z5>;
for> end for;
 <1, 0, 1, 0, (1 2 2 3 2 1)>
 <2, 1, 0, (1 1 2 3 2 1), 0>
 <3, 1, 0, (1 1 2 2 2 1), 0>
 <4, 0, 1, 0, (1 1 2 2 1 1)>
 <5, 1, 0, (1 1 1 2 2 1), 0>
 <6, 0, 1, 0, (1 1 1 2 1 1)>
 <7, 0, 1, 0, (1 1 1 1 1 1)>
 <8, 1, 0, (1 0 1 1 1 1), 0>
```

From this calculation we see that

$$\xi(z_4 - z_5) \star z_4 = \frac{1}{2}\xi\lambda(\mu+1)z_5 - \frac{1}{2}\lambda(\mu-1)z_4.$$

Hence for $\xi(z_4 - z_5)$ to be an identity element (on the left) we must have $\mu = -1$ and $\xi\lambda = 1$. And now $\xi(z_4 - z_5) \star z_i = z_i$ for all i.

Next we determine the conditions for $\xi(z_4 - z_5)$ to be an identity on the right.

```
> for i := 1 to 8 do
for> c4 := LieStructureConstant(W,X[i],Y[4]);
for> c5 := LieStructureConstant(W,X[i],Y[5]);
for> z4 := c4 eq 0 select 0 else X[i]+Y[4];
for> z5 := c5 eq 0 select 0 else X[i]+Y[5];
for> <i,c4,c5,z4,z5>;
for> end for;
 <1,  0,  1, 0, (1 2 2 3 2 1)>
 <2, -1,  0, (1 1 2 3 2 1), 0>
 <3, -1,  0, (1 1 2 2 2 1), 0>
 <4,  0,  1, 0, (1 1 1 2 2 1)>
 <5,  1,  0, (1 1 2 2 1 1), 0>
 <6,  0, -1, 0, (1 1 1 2 1 1)>
 <7,  0,  1, 0, (1 1 1 1 1 1)>
 <8,  1,  0, (1 0 1 1 1 1), 0>
```

Thus

$$z_4 * \xi(z_4 - z_5) = \tfrac{1}{2}(\lambda + 1)z_4 - \tfrac{1}{2}(\lambda - 1)z_5$$

and so $\lambda = \xi = 1$.

The revised values of P and Q are

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \quad Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Here is the MAGMAcode for these matrices.

```
> M := MatrixRing(RationalField(),8);
> P := M!1;
> P[5,5] := -1;
> P[6,6] := -1;
> P[8,8] := -1;
> Q := M!1;
> Q[4,4] := 0;
> Q[4,5] := 1;
> Q[5,4] := 1;
> Q[5,5] := 0;
```

We can check that the representations of G_2 on V_x, V_y and V_z are the isomorphic:

```
> [P*x*P^-1 : x in eX ] eq eZ;
true
> [P*x*P^-1 : x in fX ] eq fZ;
true
> [Q*x*Q^-1 : x in eY ] eq eZ;
true
> [Q*x*Q^-1 : x in fY ] eq fZ;
true
```

The product $z_i \star z_j$ is cz_k, where c is `coeff[i,j]` and k is `ndx[i,j]`.

```
> coeff := M!0;
> ndx := M!0;
> perm := [1,2,3,5,4,6,7,8];
> sgn := [1,1,1,1,-1,-1,1,-1];
> for i := 1 to 8 do for j := 1 to 8 do
for|for> c := LieStructureConstant(W,X[i],Y[perm[j]]);
for|for> if c ne 0 then
for|for|if> ndx[i,j] := Index(Z,X[i]+Y[perm[j]]);
for|for|if> coeff[i,j] := c*sgn[i];
for|for|if> end if;
for|for> end for; end for;
>
```

The Lie algebra G_2 acts on E_6 as derivations and this carries over to its action on the algebra V. In order to check this we use the following code, utilising the variables `V`, `B`, `coeff` and `ndx` already defined

```
> omult := func< u, v |
> &+[u[i]*v[j]*coeff[i,j]*B[k] : i,j in [1..8] | k ne 0
> where k is ndx[i,j]] >;
```

To check that each basis element of G_2 is a derivation of V:

```
> for g in [eZ[1],eZ[2],fZ[1],fZ[2]] do
for> forall{<i,j> : i,j in [1..8] |
for> omult(B[i],B[j])*g eq omult(B[i]*g,B[j])+omult(B[i],B[j]*g)};
for> end for;
 true
 true
 true
 true
```

Check the alternative law on the basis elements

```
> forall{<i,j> : i,j in [1..8] |
> omult(omult(B[i],B[i]),B[j]) eq omult(B[i],omult(B[i],B[j]))};
true
> forall{<i,j> : i,j in [1..8] |
> omult(omult(B[i],B[j]),B[j]) eq omult(B[i],omult(B[j],B[j]))};
true
```

The algebra is not associative:

```
> exists(i,j,k){<i,j,k> : i,j,k in [1..8] |
> omult(omult(B[i],B[j]),B[k]) ne omult(B[i],omult(B[j],B[k]))};
true
> i,j,k;
1 2 7
```

The structure constants of this algebra are all integers and so the $\mathbf{Z}$-span $\mathbf{O}(\mathbf{Z})$ of the basis z_1, $\ldots$, z_8 is an algebra. Then for any field $\mathbf{F}$ the vector space $\mathbf{O}(\mathbf{F}) = \mathbf{F} \otimes_{\mathbf{Z}} \mathbf{O}(\mathbf{Z})$ is the algebra of split octonions over $\mathbf{F}$. Our construction shows that it admits the Chevalley group $G_2(\mathbf{F})$ as a group of automorphisms.

Chapter 15

Invariant Rings of Finite Groups

15.1 Constructing Invariants

A module for constructing both characteristic zero and modular invariants of finite groups has been developed by Gregor Kemper and Allan Steel. This includes a new algorithm for computing primary invariants that guarantees that the degrees of the invariants constructed are optimal (with respect to their product and then sum). Magma allows computation in invariant rings over ground fields of arbitrary characteristic. Of particular interest is the *modular* case, i.e., the case where the characteristic of the ground field divides the order of the group.

- Invariant ring structure of a finite matrix or permutation group
- Independent homogeneous invariants of a specific degree
- Molien series
- Primary invariants having optimal degrees (with respect to their product and then sum)
- Secondary invariants of optimal degrees
- Fundamental invariants
- Algebraic relations between invariants
- Hilbert series
- Free resolution, depth and homological dimension
- Properties: Polynomial ring, Cohen-Macaulay ring

15.1.1 The Fundamental Invariants of the Degree-6 Dihedral Group

In this example we let R be the invariant ring of the degree-6 permutation representation of the dihedral group of order 12 and with coefficient field the rational field $\mathbb{Q}$.

We first compute primary invariants and secondary invariants of R easily. The secondary invariants are quite large and messy so we only print their degrees.

```
> G := DihedralGroup(6);
> G;
Permutation group G acting on a set of cardinality 6
Order = 12 = 2^2 * 3
    (1, 2, 3, 4, 5, 6)
    (1, 6)(2, 5)(3, 4)
```

```
> R := InvariantRing(G, RationalField());

> time PrimaryInvariants(R);
[
    x1 + x2 + x3 + x4 + x5 + x6,
    x1^2 + x2^2 + x3^2 + x4^2 + x5^2 + x6^2,
    x1*x2 + x1*x6 + x2*x3 + x3*x4 + x4*x5 + x5*x6,
    x1*x3 + x1*x5 + x2*x4 + x2*x6 + x3*x5 + x4*x6,
    x1^3 + x2^3 + x3^3 + x4^3 + x5^3 + x6^3,
    x1^6 + x2^6 + x3^6 + x4^6 + x5^6 + x6^6
]
Time: 0.310

> time S := SecondaryInvariants(R);
Time: 0.990
> #S;
12
> [TotalDegree(f): f in S];
[ 0, 3, 3, 4, 4, 4, 5, 6, 7, 7, 8, 9 ]
```

Now we compute fundamental invariants F of R, which generate R as an algebra over $\mathbb{Q}$. Thus any invariant of the group G can be written as an (algebraic) expression in the elements of F.

```
> time F := FundamentalInvariants(R);
Time: 0.410
> [TotalDegree(f): f in F];
[ 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 6 ]
> F;
[
    x1 + x2 + x3 + x4 + x5 + x6,
    x1^2 + x2^2 + x3^2 + x4^2 + x5^2 + x6^2,
    x1*x2 + x1*x6 + x2*x3 + x3*x4 + x4*x5 + x5*x6,
    x1*x3 + x1*x5 + x2*x4 + x2*x6 + x3*x5 + x4*x6,
    x1^3 + x2^3 + x3^3 + x4^3 + x5^3 + x6^3,
    x1^2*x2 + x1^2*x6 + x1*x2^2 + x1*x6^2 + x2^2*x3 + x2*x3^2 + x3^2*x4 +
        x3*x4^2 + x4^2*x5 + x4*x5^2 + x5^2*x6 + x5*x6^2,
    x1^2*x3 + x1^2*x5 + x1*x3^2 + x1*x5^2 + x2^2*x4 + x2^2*x6 + x2*x4^2 +
        x2*x6^2 + x3^2*x5 + x3*x5^2 + x4^2*x6 + x4*x6^2,
    x1^4 + x2^4 + x3^4 + x4^4 + x5^4 + x6^4,
    x1^3*x2 + x1^3*x6 + x1*x2^3 + x1*x6^3 + x2^3*x3 + x2*x3^3 + x3^3*x4 +
        x3*x4^3 + x4^3*x5 + x4*x5^3 + x5^3*x6 + x5*x6^3,
    x1^2*x2^2 + x1^2*x6^2 + x2^2*x3^2 + x3^2*x4^2 + x4^2*x5^2 + x5^2*x6^2,
    x1^5 + x2^5 + x3^5 + x4^5 + x5^5 + x6^5,
    x1^6 + x2^6 + x3^6 + x4^6 + x5^6 + x6^6
]
```

15.1.2 The Primary Invariants of a 4-dimensional Reflection Group

In this example we define a 4-dimensional reflection group G of order 92160 and "manually" find primary invariants for the ring of invariants of G. The function `PrimaryInvariants` would do all this automatically for us of course.

We start by constructing the Molien series of G, both as a rational function and as a power series; from this, it emerges that possible primary invariants have degrees 8, 24, 24, and 40. Next, we construct linearly independent invariants of these degrees. We then use the *Hilbert-driven Buchberger* algorithm to show that the ideal generated by these invariants is zero-dimensional, since the numerator of the Hilbert Series of the ideal is $(1-t^8)(1-t^{24})(1-t^{24})(1-t^{40})$ as expected. Since the ideal is zero-dimensional, the four invariants must be primary invariants for G.

We create the cyclotomic field $K = \mathbb{Q}(\zeta_8)$, and a matrix group G over K generated by 4 matrices:

```
> K<zeta> := CyclotomicField(8);
> G := MatrixGroup<4, K |
>             [ h, h, h, h,
>               h,-h, h,-h,
>               h, h,-h,-h,
>               h,-h,-h, h ] where h is 1/2,
>
>             [-1, 0, 0, 0,
>               0, 1, 0, 0,
>               0, 0, 1, 0,
>               0, 0, 0, 1 ],
>
>             [ 1, 0, 0, 0,
>               0, w, 0, 0,
>               0, 0, 1, 0,
>               0, 0, 0, w ] where w is zeta^2,
>
>             [ l, 0, 0, 0,
>               0, l, 0, 0,
>               0, 0, l, 0,
>               0, 0, 0, l ] where l is zeta >;

> Order(G);
92160
> FactoredOrder(G);
[ <2, 11>, <3, 2>, <5, 1> ]
```

The Molien series calculation needs the conjugacy classes of G. We call the **Classes** function, specifying that the classes C are to be found as orbits under conjugation action. We do this since the default method would be much slower for this group.

```
> C := Classes(G: Al := "Action");
> #C;
118
```

Once the conjugacy classes of G have been found, they will be remembered for subsequent calculations.

We are now in a position to compute the Molien series of G, initially as a rational function and then as a power series:

```
> MS<t> := MolienSeries(G);
> MS;
```

```
(t^32 + 1)/(t^96 - t^88 - 2*t^72 + 2*t^64 - t^56 + 2*t^48
    - t^40 + 2*t^32 - 2*t^24 - t^8 + 1)

> P<x> := PowerSeriesRing(IntegerRing(), 200);
> P ! MS;
1 + x^8 + x^16 + 3*x^24 + 4*x^32 + 5*x^40 + 8*x^48 +
    10*x^56 + 12*x^64 + 17*x^72 + 21*x^80 + 24*x^88 +
    31*x^96 + 37*x^104 + 42*x^112 + 52*x^120 + 60*x^128 +
    67*x^136 + 80*x^144 + 91*x^152 + 101*x^160 + 117*x^168
    + 131*x^176 + 144*x^184 + 164*x^192 + O(x^200)
```

It is known that the Molien series can be written in the form

$$\frac{1+t^m}{(1-t^{s_1})(1-t^{s_2})\cdots(1-t^{s_d})}$$

where d is the degree of the matrix representation. Therefore we undertake a partial factorization of the denominator D as the product of polynomials of the form $(1-x^k)$ for various k. We determine each k by taking the degree of the first non-constant term of D and then dividing out by $(1-x^k)$.

```
> D<t> := Denominator(MS);
> F := D div (1 - t^8);
> F;
-t^88 + 2*t^64 + t^48 - t^40 - 2*t^24 + 1
> F := F div (1 - t^24);
> F;
t^64 - t^40 - t^24 + 1
> F := F div (1 - t^24);
> F;
-t^40 + 1
> // Check the product
> D eq (1 - t^8)*(1 - t^24)*(1 - t^24)*(1 - t^40);
true
```

So the Molien series for G may be written as

$$\frac{1+t^{32}}{(1-t^8)(1-t^{24})(1-t^{24})(1-t^{40})}.$$

Therefore the degrees 8, 24, 24 and 40 should be tried to find primary invariants.

We now proceed to form linearly independent invariants with these degrees. The first invariant can be computed with the function **ReynoldsOperator**, and the others can be constructed using **InvariantsOfDegree**: which successively calls **ReynoldsOperator** efficiently to obtain linearly independent invariants of the desired degree.

```
> P<x1, x2, x3, x4> := PolynomialRing(K, 4);
> time i8 := ReynoldsOperator(x1^8, G);
Time: 69.980
> i8;
1/32*x1^8 + 7/16*x1^4*x2^4 + 7/16*x1^4*x3^4 + 7/16*x1^4*x4^4
    + 21/4*x1^2*x2^2*x3^2*x4^2 + 1/32*x2^8 + 7/16*x2^4*x3^4
```

```
    + 7/16*x2^4*x4^4 + 1/32*x3^8 + 7/16*x3^4*x4^4 +
    1/32*x4^8
> L8 := [32 * i8];
> time L24 := InvariantsOfDegree(G, P, 24, 2);
Time: 221.849
> time L40 := InvariantsOfDegree(G, P, 40, 1);
Time: 1591.239
> L := L8 cat L24 cat L40;
> L;
[
    x1^8 + 14*x1^4*x2^4 + 14*x1^4*x3^4 + 14*x1^4*x4^4 +
        168*x1^2*x2^2*x3^2*x4^2 + x2^8 + 14*x2^4*x3^4 +
        14*x2^4*x4^4 + x3^8 + 14*x3^4*x4^4 + x4^8,
    x1^24 - 35420/771*x1^18*x2^2*x3^2*x4^2 +
            ...
        + 2576*x3^12*x4^12 + 759*x3^8*x4^16 + x4^24,
    x1^20*x2^4 + x1^20*x3^4 + x1^20*x4^4 +
            ...
        + x3^4*x4^20,
    x1^40 + 38/109*x1^36*x2^4 + 38/109*x1^36*x3^4 +
            ...
        38/109*x3^4*x4^36 + x4^40
]
```

(The output above has been heavily edited, since the polynomials have many terms.)

Finally, we show that the ideal is zero-dimensional by checking that the Hilbert-driven Buchberger algorithm succeeds:

```
> time b, h := HilbertGroebnerBasis(L, D);
Time: 3979.420
> b;
true
```

This indicates that the ideal is zero-dimensional, so L must contain primary invariants for G.

15.2 Properties of Invariant Rings

- Algebraic relations between invariants
- Hilbert series
- Free resolution
- Depth and homological dimension
- Properties of an invariant ring: Polynomial ring, Cohen-Macaulay ring

15.2.1 Invariant Ring of the Degree 5 Jordan Block

Let R be the invariant ring of the group generated by the degree-5 Jordan block over GF(2). We first note that R is not Cohen-Macaulay. We then construct a minimal free resolution F of R and verify that the homological dimension of R is 2 so the depth of R is 3.

```
> K:=GF(2);
> G := MatrixGroup<5,K | [1,0,0,0,0, 1,1,0,0,0, 0,1,1,0,0,
>                         0,0,1,1,0, 0,0,0,1,1]>;
> Order(G);
8
> G;
MatrixGroup(5, GF(2)) of order 2^3
Generators:
    [1 0 0 0 0]
    [1 1 0 0 0]
    [0 1 1 0 0]
    [0 0 1 1 0]
    [0 0 0 1 1]
> R := InvariantRing(G);
```

We now compute the primary and secondary invariants and list their degrees.

```
> [TotalDegree(f): f in PrimaryInvariants(R)];
[ 1, 2, 2, 4, 8 ]

> [TotalDegree(f): f in SecondaryInvariants(R)];
[ 0, 3, 3, 3, 4, 4, 5, 5, 5, 6, 6, 6, 7, 7, 7, 7, 8, 8, 9, 9, 10, 11 ]
```

We check whether R is Cohen-Macaulay.

```
> IsCohenMacaulay(R);
false

> HilbertSeries(R);
(-t^6 + t^4 - t^3 - 2*t^2 + 2*t - 1)/(t^13 - 3*t^12 + 2*t^11 +
   2*t^10 - 3*t^9 + t^8 - t^5 + 3*t^4 - 2*t^3 - 2*t^2 + 3*t - 1)
```

Next we construct a minimal free resolution for R.

```
> time F := MinimalFreeResolution(R);
Time: 14.829
> #F;
3
> HomologicalDimension(R);
2
> Depth(R);
3
```

Note that since the homological dimension of R is 2, the depth of R has to be 3.

15.2.2 The Invariant Ring of a Matrix Group and its Dual

We take a subgroup of order 16 of $\mathrm{GL}_7(2)$ and compute its invariant ring.

```
> K:=GF(2);
> G:=MatrixGroup<7,K |
>  [1,0,0,0,1,0,0, 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, 0,0,0,0,0,1,0, 0,0,0,0,0,0,1],
>  [1,0,0,0,0,0,0, 0,1,0,0,0,1,0, 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, 0,0,0,0,0,0,1],
>  [1,0,0,0,0,0,0, 0,1,0,0,0,0,0, 0,0,1,0,0,0,1, 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],
>  [1,0,0,0,0,0,0, 0,1,0,0,0,0,0, 0,0,1,0,0,0,0, 0,0,0,1,1,1,1,
>   0,0,0,0,1,0,0, 0,0,0,0,0,1,0, 0,0,0,0,0,0,1] >;
> #G;
16
> R:=InvariantRing(G);
> time SecondaryInvariants(R);
[
   1
]
> IsCohenMacaulay(R);
true
Time: 0.270
```

Since R has only one secondary invariant, it is (isomorphic to) a polynomial ring and thus Cohen-Macaulay.

Now we take a look at the dual group's invariants.

```
> H:=MatrixGroup<7,K | [Transpose(g): g in Generators(G)]>;
> R:=InvariantRing(H);
> time IsCohenMacaulay(R);
false
Time: 12.279
> Depth(R);
6
> HilbertSeries(R);
(-t^5 + t^4 - 3*t^3 - 1)/(t^16 - 4*t^15 + 6*t^14 - 4*t^13 - 2*t^12 + 12*t^11 -
18*t^10 + 12*t^9 - 12*t^7 + 18*t^6 - 12*t^5 + 2*t^4 + 4*t^3 - 6*t^2 + 4*t - 1)
```

The dual group's invariant ring is not even Cohen-Macaulay!

Chapter 16

Vector Spaces and KG-Modules

16.1 Introduction

The four fundamental algorithms for computational module theory are echelonization, the spinning algorithm, the Meataxe algorithm and an algorithm for $\mathrm{Hom}(U, V)$. For the important case of tuple modules over finite fields, different representations of vector arithmetic, depending upon the field, have been implemented. The Magma algorithm for splitting modules (the Meataxe algorithm) is a deterministic version of the Holt-Rees algorithm and is capable of splitting modules over GF(2) having dimension up to at least 20 000. While the Magma Meataxe currently works over finite fields, it is being extended to modules defined over $\mathbb{Q}$, or small degree extensions of $\mathbb{Q}$. Magma also includes a new algorithm for the construction of $\mathrm{Hom}(U, V)$ where U and V are KG-modules which is applicable to modules having dimension several hundred.

16.1.1 General tuple modules over fields

- Arithmetic
- Extension and restriction of the field of scalars
- Direct sum, tensor product, symmetric square, exterior square
- Submodules, via the spinning algorithm
- Membership of a submodule
- Basis operations
- Sum and intersection of submodules
- Quotient modules
- Splitting a reducible module (Holt-Rees Meataxe)
- Testing a module for irreducibility, absolute irreducibility
- Centralizing algebra of an irreducible module
- Composition series, composition factors, constituents
- Maximal and minimal submodules, Jacobson radical, socle
- Socle series
- Construction of $\mathrm{Hom}(U, V)$, $\mathrm{End}(U)$
- Testing modules for isomorphism
- Complement of a direct summand
- Testing modules for indecomposability; indecomposable components
- Submodule lattice (modules over a finite field)

16.1.2 KG-Modules

- As for modules over a field
- Dual
- Construction of a permutation module
- Induction and restriction

16.2 Composition factors of a permutation module

We load the simple Lie group $G(2,5)$, represented as a permutation group of degree 3096, from the standard Magma library of algebraic structures. Then we find the composition factors of a composition series of the permutation module of G:

```
> load "g25";
> P := PermutationModule(G, GF(2));
> time CompositionFactors(P);
[
    GModule of dimension 650 over GF(2),
    GModule of dimension 280 over GF(2),
    GModule of dimension 1240 over GF(2),
    GModule of dimension 650 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 1084 over GF(2),
    GModule of dimension 1 over GF(2)
]
Time: 345.759
```

16.3 Constituents of a module

Construct a permutation group G.

```
> G := PermutationGroup<20 |
>     (1, 6, 11, 16)(2, 7, 12, 17)(3, 8, 13, 18)(4, 9, 14, 19)(5, 10, 15, 20),
>     (1, 6)(2, 7)(3, 8)(4, 9)(5, 10),
>     (1, 2, 3, 4, 5),
>     (1, 5)(2, 4)
> >;
```

Construct the permutation module of G over GF(7).

```
> P := PermutationModule(G, GF(7));
> P;
GModule P of dimension 20 with base ring GF(7)
>
> // Find the Composition factors of P.
> F := CompositionFactors(P);
```

```
[
    GModule of dimension 1 with base ring GF(7),
    GModule of dimension 3 with base ring GF(7),
    GModule of dimension 16 with base ring GF(7)
]
```

Form a tensor product.

```
> T := TensorProduct(F[3], F[3]);
> T;
GModule T of dimension 256 with base ring GF(7)
>
> CompositionFactors(T);
[
    GModule of dimension 48 with base ring GF(7),
    GModule of dimension 1 with base ring GF(7),
    GModule of dimension 16 with base ring GF(7),
    GModule of dimension 1 with base ring GF(7),
    GModule of dimension 48 with base ring GF(7),
    GModule of dimension 3 with base ring GF(7),
    GModule of dimension 48 with base ring GF(7),
    GModule of dimension 4 with base ring GF(7),
    GModule of dimension 4 with base ring GF(7),
    GModule of dimension 16 with base ring GF(7),
    GModule of dimension 16 with base ring GF(7),
    GModule of dimension 48 with base ring GF(7),
    GModule of dimension 3 with base ring GF(7)
]
```

Classify the modules up to isomorphism and select one module from each isomorphism class.

```
> Constituents(T);
[
    GModule of dimension 1 with base ring GF(7),
    GModule of dimension 3 with base ring GF(7),
    GModule of dimension 4 with base ring GF(7),
    GModule of dimension 16 with base ring GF(7),
    GModule of dimension 48 with base ring GF(7),
    GModule of dimension 48 with base ring GF(7),
    GModule of dimension 48 with base ring GF(7)
]
```

16.4 Constructing an endo-trivial module

This example is due to Jon Carlson (Athens, GA, USA). The idea is to test a technique for constructing endo-trivial modules. An endo-trivial module is one with the property that

$$Hom_k(M, M) = M \otimes Dual(M)$$

is the direct sum of a trivial module and a projective (free, in this case) module.

First we construct an extraspecial group of order 27 and exponent 3.

```
> ps := PSL(3, 3);
> ps;
Permutation group ps acting on a set of cardinality 13
    (1, 10, 4)(6, 9, 7)(8, 12, 13)
    (1, 3, 2)(4, 9, 5)(7, 8, 12)(10, 13, 11)
> g := SylowSubgroup(ps, 3);
> g;
Permutation group g acting on a set of cardinality 13
Order = 27 = 3^3
    (3, 13, 9)(5, 8, 6)(7, 11, 12)
    (2, 5, 3)(4, 8, 9)(6, 13, 10)
```

Now we create the module in question. It is the kernel δ_x in an exact sequence

$$0 \to \delta_x \to x \to k \to 0$$

where k is the trivial $f3[g]$-module and x is a permutation module whose point stabilizer is a noncentral cyclic subgroup.

```
> g.1 in Centre(g);
false
> h := sub<g | g.1>;
> h;
Permutation group h acting on a set of cardinality 13
    (2, 10, 4)(5, 8, 6)(7, 12, 11)
> F3 := GaloisField(3);
> x := PermutationModule(g, h, F3);
> hhh := GHom(x, TrivialModule(g, F3));
> hhh;
KMatrixSpace of 9 by 1 GHom matrices and dimension 1 over GF(3)

> delx := Kernel(hhh.1);
> delx;
GModule delx of dimension 8 over GF(3)
> xx := TensorProduct(delx, delx);
> xx;
GModule xx of dimension 64 over GF(3)
```

Now we want to decompose the tensor product of δ_x with itself. One of the summands should be an endo-trivial module. Note that the dimension of an endo-trivial module cannot be divisible by the prime 3, since the square of the dimension must be 1 plus a multiple of 27 (the order of the group g). The function `IsDecomposable` tests whether its argument is decomposable, and if this is the case then it also provides a decomposition as the second and third return values.

```
> a, m1, m2 := IsDecomposable(xx);
> a;
true
> m1, m2;
GModule m1 of dimension 9 over GF(3)
GModule m2 of dimension 55 over GF(3)
```

We want to check what the pieces are. We suspect that the module of dimension 9 is just a copy of our permutation module, and the check below confirms that. Then we proceed with the other piece.

```
> IsIsomorphic(m1, x);
true

> a,m3,m4 := IsDecomposable(m2);
> a, m3, m4;
true
GModule m3 of dimension 27 over GF(3)
GModule m4 of dimension 28 over GF(3)
```

We suspect this time that the module of dimension 27 is a free module. We use the theorem that the free module is the only module with the property that it is generated by a single element and has dimension equal to the order of the group. So we try a couple of times to see if it can be generated by a single element.

```
> sub< m3 | Random(m3) >;
GModule of dimension 21 over GF(3)
> sub< m3 | Random(m3) >;
GModule m3 of dimension 27 over GF(3)
```

So $m3$ is a free module. We can proceed.

```
> IsDecomposable(m4);
false
```

Now we check to see if $m4$ is endo-trivial.

```
> et := TensorProduct(m4, Dual(m4));
> et;
GModule et of dimension 784 over GF(3)
> Quotrem(Dimension(et), #g);
29 1
```

So the dimension is 1 more than a multiple (29) of the order of g (27), as expected.

We know that the tensor product of $m4$ with its dual has a direct summand isomorphic to the trivial module. If it is endo-trivial then the tensor of it with its dual must be one copy of the trivial module plus $(\mathrm{Dim}(et) - 1)/27 = 29$ copies of the free module. So the action of the group algebra must have exactly $29 + 1 = 30$ fixed points. We check:

```
> Fix(et);
GModule of dimension 30 over GF(3)
```

Actually at this point we can be certain that $m4$ is an endo-trivial module. But just to be sure we factor out projective modules to see if we get down to the trivial module. We are using here the fact that the group ring is self-injective and hence any free submodule (module of dimension 27 generated by one element) is a direct summand.

```
> ww := et;
> Dim := Dimension; // shorthand
> repeat
>     sum := rep{s : i in [1..100] | Dim(s) eq 27
>       where s is sub< ww | Random(ww) >};
>     qq := quo< ww | sum >;
>     print (Dim(et) - Dim(qq)) / #g, Dim(qq);
>     ww := qq;
> until Dim(qq) eq 1;
1 757
2 730
3 703
4 676
   [ etc ]
26 82
27 55
28 28
29 1
```

Finally we want to check that the module $m4$ is not one of the known endo-trivial modules. It is enough to see that it doesn't have the same restriction to all of the maximal elementary abelian subgroups. So we calculate all the maximal elementary abelian 3-subgroups. and then check the dimension of the fixed point set on each.

```
> cc := Centre(g);
> max1:= sub< g | g.1, cc >;
> max2:= sub< g | g.2, cc >;
> max3:= sub< g | g.1*g.2, cc >;
> max4:= sub< g | g.1*g.2^2, cc >;
> [ Fix(Restriction(m4, x)) : x in [max1, max2, max3, max4] ];
[
    GModule of dimension 6 over GF(3),
    GModule of dimension 4 over GF(3),
    GModule of dimension 4 over GF(3),
    GModule of dimension 4 over GF(3)
]
```

Notice that the single fixed-point space of dimension 6 corresponds to the restriction of $m4$ to the maximal subgroup containing the subgroup h with which we started.

Chapter 17

Homomorphisms of Modules

17.1 Introduction

Magma provides many facilities for computing in modules of (rectangular) matrices. These often arise as the modules $\mathrm{Hom}(U, V)$ for tuple modules U and V.

- Explicit construction of $\mathrm{Hom}(U, V)$ for proper subspaces U and V
- Explicit construction of $\mathrm{Hom}(H_1, H_2)$ for homomorphism modules H_1 and H_2 with left or right matrix action
- Construction of the reduced module of a homomorphism module whose elements are with respect to the bases of the domain and codomain (not just the generic bases of these)
- Arithmetic
- Extension and restriction of the ring of scalars
- Construction of submodules, quotient modules
- Sum and intersection of submodules
- Basis operations
- Row and column operations
- Echelon form (over a field)
- Hermite and Smith normal forms (over an ED)
- Solution of systems of linear equations
- Image, kernel, cokernel
- Arithmetic
- Extension and restriction of the ring of scalars
- Construction of submodules, quotient modules
- Sum and intersection of submodules
- Hermite and Smith normal forms (LLL-based algorithms in the case of $\mathbf{Z}$-modules)
- Modules with torsion
- LLL algorithm for a basis matrix or Gram matrix

17.2 Homomorphisms between *Hom*-modules

We construct two homomorphism modules H_1 and H_2 over $\mathbb{Q}$ and then the homomorphism module $H = \mathrm{Hom}(H_1, H_2)$ with right matrix action.

```
> Q := RationalField();
> H1 := sub<RMatrixSpace(Q, 2, 3) | [1,2,3, 4,5,6], [0,0,1, 1,3,3]>;
> H2 := sub<RMatrixSpace(Q, 2, 4) | [6,5,7,1, 15,14,16,4], [0,0,0,0, 1,2,3,4]>;
> H := Hom(H1, H2, "right");
> H: Maximal;
KMatrixSpace of 3 by 4 matrices and dimension 1 over Rational Field
Echelonized basis:

[   1    2    3    4]
[-1/2   -1 -3/2   -2]
[   0    0    0    0]

> H1.1 * H.1;
[  0   0   0   0]
[3/2   3 9/2   6]
> H1.1 * H.1 in H2;
true
> Image(H.1): Maximal;
KMatrixSpace of 2 by 4 matrices and dimension 1 over Rational Field
Echelonized basis:
\bln
[0 0 0 0]
[1 2 3 4]

> Kernel(H.1): Maximal;
KMatrixSpace of 2 by 3 matrices and dimension 1 over Rational Field
Echelonized basis:

[ 1  2  6]
[ 7 14 15]

> H1 := sub<RMatrixSpace(Q,2,3) | [1,2,3, 4,5,6]>;
> H2 := sub<RMatrixSpace(Q,3,3) | [1,2,3, 5,7,9, 4,5,6]>;
> H := Hom(H1, H2, "left");
> H: Maximal;
KMatrixSpace of 3 by 2 matrices and dimension 1 over Rational Field
Echelonized basis:

[1 0]
[1 1]
[0 1]
> Image(H.1);
KMatrixSpace of 3 by 3 matrices and dimension 1 over Rational Field
> Kernel(H.1);
KMatrixSpace of 2 by 3 matrices and dimension 0 over Rational Field
```

17.3 Smith form of integer matrices

Here we give 3 applications of the Smith form algorithm.

(1) Set *HS* to be the Higman-Sims simple group:

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

Construct the orbital graph G with 100 vertices, and set A to be the 100 by 100 adjacency matrix of graph G:

```
> G := OrbitalGraph(HS, 1, {2});
> A := AdjacencyMatrix(G);
> Parent(A);
Full Matrix Algebra of degree 100 over Integer Ring
```

Note that A has 7700 1-entries (out of a total of 10000 entries):

```
> &+Eltseq(A);
7700
```

Print the elementary divisors of A using the Smith form algorithm:

```
> time ElementaryDivisors(A);
[ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21,
21, 21, 21, 231 ]
Time: 0.200
```

Of course, one can calculate the determinant also using the Smith algorithm:

```
> time Determinant(A);
-164810250280066269858882875801738539817056953234887738979 9
Time: 0.200
```

(2) Compute the abelian-quotient invariants of a normal subgroup of $G(7,8)$ using the Smith algorithm.

```
> G := func< alpha, n |
>     Group<a, b, d, e | a^2 = b^n =
>     a * b^-1 * a * b * e^(alpha - 1) * a * b^2 * a * b^-2 = 1,
>     e = a * b * a * b^-1, d^b = e,
>     (e * b)^n = (d, e^(alpha - 1)) = 1, e^b = e^alpha * d> >;
> G7_8<a, b, d, e> := G(7, 8);
> R := ncl<G7_8 | e^3>;
> k := Rewrite(G7_8, R: Simplify := false);
```

Apply the Smith algorithm to the relevant matrix with 1081 rows and 2657 columns.

```
> time AbelianQuotientInvariants(k);
[ 2, 2, 2, 2, 102, 37842 ]
Time: 69.070
```

(3) Calculating the Smith form or determinant of arbitrary matrices is easy too:

```
> m := MatrixAlgebra(IntegerRing(), 50);
> a := m ! [Random(0,1): i in [1..50^2]];
> a[1];  // the first row
(1 1 0 0 0 1 1 1 1 0 0 1 0 1 0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 0 0 0
1 0 0 0 1 0 1 1 1 1)
> time ElementaryDivisors(a);
[ 2, 87285028673206080 ]
Time: 0.380
```

17.4 Solution of matrix equations

We construct a 10 by 10 matrix X of integers.

```
> M := MatrixRing(IntegerRing(), 10);
> M;
Full Matrix Algebra of degree 10 over Integer Ring
> X := M ! [i+2*j: i,j in [1..10]];
> X;
[ 3  5  7  9 11 13 15 17 19 21]
[ 4  6  8 10 12 14 16 18 20 22]
[ 5  7  9 11 13 15 17 19 21 23]
[ 6  8 10 12 14 16 18 20 22 24]
[ 7  9 11 13 15 17 19 21 23 25]
[ 8 10 12 14 16 18 20 22 24 26]
```

```
[ 9 11 13 15 17 19 21 23 25 27]
[10 12 14 16 18 20 22 24 26 28]
[11 13 15 17 19 21 23 25 27 29]
[12 14 16 18 20 22 24 26 28 30]
```

We set V to be the length 10 vector $(5, 4, 3, 5, 6, 4, 3, 7, 6, 5)$ and then set W to be VX.

```
> V := RSpace(IntegerRing(), 10) ! [5, 4, 3, 5, 6, 4, 3, 7, 6, 5];
> V;
(5 4 3 5 6 4 3 7 6 5)
> W := V * X;
> W;
( 373  469  565  661  757  853  949 1045 1141 1237)
```

Now we find a solution S of the equation $SX = W$ together with the kernel K of X.

```
> S, K := Solution(X, W);
> S;
(-181  229    0    0    0    0    0    0    0    0)
> K;
RSpace of degree 10, dimension 8 over Integer Ring
Echelonized basis:
( 1  0  0  0  0  0  0  0 -9  8)
( 0  1  0  0  0  0  0  0 -8  7)
( 0  0  1  0  0  0  0  0 -7  6)
( 0  0  0  1  0  0  0  0 -6  5)
( 0  0  0  0  1  0  0  0 -5  4)
( 0  0  0  0  0  1  0  0 -4  3)
( 0  0  0  0  0  0  1  0 -3  2)
( 0  0  0  0  0  0  0  1 -2  1)
```

Finally we note that $V - S$ is in the kernel as expected.

```
> V - s in K;
true
```

Chapter 18

Lattices

18.1 Introduction

A lattice in Magma is a $\mathbf{Z}$-module contained in $\mathbf{R}^n$ with some additional structure, in particular an inner product. The basic information for a lattice is a basis, given by a sequence of vectors in $\mathbf{R}^n$, and an inner product $(\cdot, \cdot)$ given by a positive definite matrix M such that $(v, w) = vMw^{tr}$. Central to the lattice machinery in Magma is a highly optimized LLL algorithm. The LLL algorithm takes a basis of a lattice and returns a new basis of the lattice which is *LLL-reduced* which usually means that the vectors of the new basis have small norms. The Magma LLL algorithm is based on the FP-LLL algorithm of Schnorr and Euchner and the de Weger integral algorithm but includes various optimizations, with particular attention to different kinds of input matrices.

18.2 Construction and Operations

- Creation of a lattice by a given generating matrix or basis matrix together with an optional inner product matrix
- Creation of a lattice by a given Gram matrix
- Construction of lattices from codes
- Construction of lattices from algebraic number fields
- Construction of special lattices, including the root lattices A_n, D_n, E_n; the laminated lattices Λ_n (including the Barnes-Wall lattice Λ_{16} and the Leech Lattice Λ_{24}); the Kappa lattices K_n, etc.
- Creation of and arithmetic with lattice elements
- Inner product, norm, and length of lattice elements with respect to the inner product of the lattice
- Conversion between a lattice element and its coordinates with respect to the basis of a lattice (in both directions)
- Action on lattice elements by matrices
- Creation of sublattices and superlattices, scaling of lattices
- Creation of quotient lattices (abelian group with isomorphism)
- Dual of a lattice, dual quotient of a lattice

- Arithmetic on lattices: sum, intersection, direct sum, tensor product, exterior square, symmetric square
- Conversion between lattices and $\mathbf{Z}$-modules and $\mathbb{Q}$-modules.

18.2.1 Example: Constructing the Barnes-Wall Lattice

The 16-dimensional Barnes-Wall lattice Λ_{16} can be constructed from the first order Reed-Muller code of length 16 using construction 'B'. Note that the inner product matrix is the identity matrix divided by 2 so that the Gram matrix is integral and primitive.

```
> C := ReedMullerCode(1, 4);
> C: Minimal;
[16, 5, 8] Reed-Muller Code (r = 1, m = 4) over GF(2)
> L := Lattice(C, "B");
> L;
Lattice of rank 16 and degree 16
Basis:
(1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1)
(0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1)
(0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1)
(0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 2)
(0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1)
(0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 2)
(0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 2)
(0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 2)
(0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1)
(0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 2)
(0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 2)
(0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 2)
(0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 2)
(0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4)
Inner Product denominator: 2
```

18.3 Properties

- Rank, determinant, basis, basis matrix, inner product matrix, Gram matrix, centre density, testing for integrality and evenness, index in a superlattice
- Minimum of a lattice (which can also be asserted)
- Kissing number of a lattice
- Theta series of a lattice
- Enumeration of all short vectors of a lattice having norm in a given range
- Enumeration of all shortest vectors of a lattice
- Enumeration of all vectors of a lattice having squared distance from a vector (possibly) outside the lattice in a given range

- Enumeration of all vectors of a lattice closest to a vector (possibly) outside the lattice
- Process to enumerate short or close vectors of a lattice thereby allowing manual looping over short vectors having norm in a given range or close vectors having squared distance in a given range
- Pure lattice of a lattice over $\mathbf{Z}$ or $\mathbb{Q}$
- Computation of a neighbour of a lattice with respect to a particular vector
- Construction of a fundamental Voronoi cell of a small-dimensional lattice
- Holes, deep holes and covering radius of a lattice
- Successive minima of a lattice
- Computation of the genus of an integral lattice

Magma includes a highly optimized algorithm for enumerating all vectors of a lattice of a given norm. This algorithm is used for computing the minimum, the shortest vectors, short vectors in a given range, and vectors close to or closest to a given vector (possibly) outside the lattice.

18.3.1 Example: Gosset Lattice

We create the Gosset lattice $L = E_8$ and find the shortest vectors of L. There are 120 normalized vectors so the kissing number is 240, and the minimum is 2.

```
> L := Lattice("E", 8);
> S := ShortestVectors(L);
> #S;
120
> KissingNumber(L);
240
> { Norm(v): v in S };
{ 2 }
> Minimum(L);
2
```

We note that the rank of the space generated by the shortest vectors is 8 so that the successive minima of L are $[2, 2, 2, 2, 2, 2, 2, 2]$.

```
> Rank(ShortestVectorsMatrix(L));
8
```

We next find the vectors in L which are closest to a certain vector in the Q-span of L. The vector is actually a hole of L and the square of its distance from L is $8/9$.

```
> w := RSpace(RationalField(), 8) !
>   [ -1/6, 1/6, -1/2, -1/6, 1/6, -1/2, 1/6, -1/2 ];
> C, d := ClosestVectors(L, w);
> C;
```

```
[
    (-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),
    (-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),
    ( 1/2  1/2 -1/2 -1/2  1/2 -1/2  1/2 -1/2),
    ( 0  0 -1  0  0 -1  0  0),
    ( 0  0 -1  0  0  0  0 -1),
    ( 0  0  0  0  0 -1  0 -1),
    (0 0 0 0 0 0 0 0)
]
> d;
8/9
> { Norm(v): v in C };
{ 0, 2 }
```

We verify that the squared distance of the vectors in C from w is $8/9$.

```
> { Norm(v - w): v in C };
{ 8/9 }
```

We finally notice that these closest vectors are in fact amongst the shortest vectors of the lattice (together with the zero vector).

```
> Set(C) subset (Set(S) join {-v: v in S} join { L!0 });
true
```

18.3.2 Example: Voronoi Cells of a Perfect Lattice

We compute the Voronoi cell of a perfect lattice of dimension 6.

```
> L := LatticeWithGram(6, [4, 1,4, 2,2,4, 2,2,1,4, 2,2,1,1,4, 2,2,2,2,2,4]);
> L;
Standard Lattice of rank 6 and degree 6
Inner Product Matrix:
[4 1 2 2 2 2]
[1 4 2 2 2 2]
[2 2 4 1 1 2]
[2 2 1 4 1 2]
[2 2 1 1 4 2]
[2 2 2 2 2 4]
> time V, E, P := VoronoiCell(L);
Time: 26.609
> #Holes(L), #DeepHoles(L), CoveringRadius(L);
782 28 5/2
```

The Voronoi cell has 782 vertices, but only 28 of these are of maximal norm $5/2$ and therefore deep holes. We now compute the norms and cardinalities for the shallow holes.

```
> M := MatrixRing(Rationals(), 6) ! InnerProductMatrix(L);
> N := [ (v*M, v) : v in V ];
> norms := Sort(Setseq(Set(N))); norms;
[ 17/9, 2, 37/18, 20/9, 7/3, 5/2 ]
> card := [ #[ x : x in N | x eq n ] : n in norms ]; card;
[ 126, 16, 288, 180, 144, 28 ]
```

So there are 126 holes of norm 17/9, 16 holes of norm 2, etc. We now investigate the Voronoi cell as a polyhedron.

```
> #V, #E, #P;
782 4074 104
> { Norm(L!p) : p in P };
{ 4, 6 }
> #ShortVectors(L, 6);
52
```

The polyhedron which is the convex closure of the holes has 782 vertices, 4074 edges and 104 faces. The faces are defined by vectors of length up to 6 and all such vectors are relevant (since there are only 104). We finally look at the graph defined by the vertices and edges of the Voronoi cell.

```
> G := VoronoiGraph(L);
> IsConnected(G);
true
> Diameter(G);
8
> Maxdeg(G);
20 ( -1   0 1/2 1/2 1/2   0)
> v := RSpace(Rationals(), 6) ! [ -1, 0, 1/2, 1/2, 1/2, 0 ]; (v*M, v);
5/2
```

The graph is (of course) connected, its diameter is 8 and the vertices of maximal degree 20 are exactly the deep holes.

18.4 Reduction

- LLL reduction of lattices, basis matrices and Gram matrices (with numerous parameters)
- Seysen reduction of lattices, basis matrices and Gram matrices (for reducing a lattice and its dual simultaneously)
- Pairwise reduction of lattices, basis matrices and Gram matrices
- Orthogonalization and orthonormalization (Cholesky decomposition) of a lattice
- Testing matrices for positive or negative (semi-)definiteness

The LLL algorithm can operate on either a basis matrix or a Gram matrix (and will use the Gram method even if given a basis matrix and it is deemed appropriate) and can be controlled by many parameters (δ constant, exact de Weger integral method or Schnorr-Euchner floating point method, step and time limits, selection of methods, etc.). The LLL algorithm can reduce matrices with very large entries as well as matrices having large sizes (e.g., number of rows well over 500).

18.4.1 Example: Knapsack Problem

Let $Q = [a_1, \ldots, a_n]$ be a sequence of (not necessarily distinct) positive integers and let s be a positive integer. We wish to find all solutions to the equation $\sum_{i=1}^{n} x_i a_i = s$ with $x_i \in \{0, 1\}$. This is known as the *Knapsack* problem. The following lattice-based solution is due to Schnorr and Euchner. To solve the problem, we create the lattice L of rank $n + 1$ and degree $n + 2$ with the following basis:

$$\begin{aligned} b_1 &= (2, 0, \ldots, 0, na_1, 0) \\ b_2 &= (0, 2, \ldots, 0, na_2, 0) \\ &\vdots \\ b_n &= (0, 0, \ldots, 2, na_n, 0) \\ b_{n+1} &= (1, 1, \ldots, 1, ns, 1). \end{aligned}$$

Then every vector $v = (v_1, \ldots, v_{n+2}) \in L$ such that the norm of v is $n + 1$ and

$$v_1, \ldots, v_n, v_{n+2} \in \{\pm 1\}, v_{n+1} = 0,$$

yields the solution $x_i = |v_i - v_{n+2}|/2$ for $i = 1, \ldots, n$ to the original equation.

We first write a function `KnapsackLattice` which, given the sequence Q and sum s, creates a matrix X representing the above basis and returns the lattice generated by the rows of X. Note that the `Lattice` creation function will automatically LLL-reduce the matrix X as it creates the lattice.

```
> function KnapsackLattice(Q, s)
>     n := #Q;
>     X := RMatrixSpace(IntegerRing(), n + 1, n + 2) ! 0;
>     for i := 1 to n do
>         X[i][i] := 2;
>         X[i][n + 1] := n * Q[i];
>         X[n + 1][i] := 1;
>     end for;
>     X[n + 1][n + 1] := n * s;
>     X[n + 1][n + 2] := 1;
>     return Lattice(X);
> end function;
```

We next write a function `Solutions` which uses the function `ShortVectors` to enumerate all vectors of the lattice L having norm exactly $n + 1$ and thus to find all solutions to the Knapsack problem associated with L. (Note that the minimum of the lattice may be less than $n + 1$.) The function returns each solution as a sequence of indices for Q.

```
> function KnapsackSolutions(L)
>     n := Rank(L) - 1;
>     M := n + 1;
>     S := ShortVectors(L, M, M);
>     return [
>         [i: i in [1 .. n] | v[i] ne v[n + 2]]: t in S |
>             forall{i: i in [1 .. n] cat [n + 2] | Abs(v[i]) eq 1} and
```

```
>                 v[n + 1] eq 0 where v is t[1]
>     ];
> end function;
```

We now apply our functions to a sequence Q of 12 integers each less than 1000 and the sum 2676. There are actually 4 solutions. We verify that each gives the original sum.

```
> Q := [ 52, 218, 755, 221, 574, 593, 172, 771, 183, 810, 437, 137 ];
> s := 2676;
> L := KnapsackLattice(Q, s);
> L;
Lattice of rank 13 and degree 14
Basis:
(  0   0   0   0   2   0   0   0   0   0  -2  -2   0   0)
(  1   1   1   1  -1  -1  -1  -1   1   1   1  -1   0  -1)
(  1   1  -1  -1  -1   1  -1  -1  -1   1   1   1   0   1)
(  0   2   0   0   0   0  -2   0  -2   0   0   2   0   0)
(  2   0   0   2   0  -2   0   0   2   0   0   2   0   0)
(  0   0   2   0   0  -2   2  -2   0   0   2   0   0   0)
(  3  -1   1   1  -1   1  -1  -1  -1  -1   1   1   0   1)
(  1   1   1  -1  -1   1   1  -1   1  -1  -1   3   0   1)
(  1  -1  -1   3  -1  -1   1   1  -1   1  -1  -1   0   1)
(  1  -1  -1   1   1   1   1  -1  -3   1   1  -1   0  -1)
(  1   3  -3   1  -1   1   1   1   1  -1   1   1   0   1)
(  1   1   1  -3   1  -1   3   1   1  -1  -1  -1   0  -1)
(  1  -1  -1   1  -1   1   1   1  -1  -1   1  -1 -12   1)
> S := KnapsackSolutions(L);
> S;
[
    [ 2, 3, 4, 5, 8, 12 ],
    [ 3, 4, 5, 7, 8, 9 ],
    [ 3, 4, 7, 8, 9, 11, 12 ],
    [ 1, 2, 3, 4, 9, 10, 11 ]
]
> [&+[Q[i]: i in s]: s in S];
[ 2676, 2676, 2676, 2676 ]
```

Finally, we apply our method to a larger example. We let Q be a sequence consisting of 50 random integers in the range $[1, 2^{1000}]$. We let I be a random subset of $\{1 \ldots 50\}$ and let s be the sum of the elements of Q indexed by I. We then solve the Knapsack problem with input (Q, s) and this time obtain I as the only answer.

```
> b := 1000;
> n := 50;
> SetSeed(1);
> Q := [Random(1, 2^b): i in [1 .. n]];
> I := {};
> while #I lt n div 2 do
>     Include(~I, Random(1, n));
> end while;
> I := Sort(Setseq(I)); I;
```

```
[ 1, 4, 5, 8, 9, 10, 13, 16, 19, 23, 24, 26, 29, 30, 31, 32, 33, 34,
35, 37, 38, 40, 42, 47, 48 ]
> s := &+[Q[i]: i in I]; Ilog2(s);
1003
> time L := KnapsackLattice(Q, s);
Time: 25.930
> [Ilog2(Norm(b)): b in Basis(L)];
[ 5, 45, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46,
46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46,
46, 46, 46, 46, 46, 46, 46, 46, 46, 47, 47, 47, 47, 47, 47, 47, 47 ]
> time KnapsackSolutions(L);
[
    [ 1, 4, 5, 8, 9, 10, 13, 16, 19, 23, 24, 26, 29, 30, 31, 32, 33,
    34, 35, 37, 38, 40, 42, 47, 48 ]
]
Time: 1.579
```

18.5 Automorphisms and G-Lattices

- Automorphism group of a lattice
- Subgroup of the automorphism group of a lattice fixing specified bilinear forms
- Determination of whether two lattices are isometric
- Determination of whether two lattices are isometric in such a way that specified bilinear forms are fixed
- Creation of G-lattices with associated operations
- Invariant lattice of a rational matrix group and the associated action (thus yielding an integral representation of the group)
- Bravais group of a finite rational matrix group
- Invariant sublattices of finite index
- Space of invariant bilinear forms
- Positive definite invariant form of a rational matrix group
- Endomorphism ring
- Centre of the endomorphism ring
- Dimension of the space of invariant bilinear forms, the endomorphism algebra or its centre using a modular algorithm

The computation of the automorphism group of a lattice and the testing of lattices for isometry is performed using the AUTO and ISOM programs of Bernd Souvignier. The automorphism group Co_0 of the 24-dimensional Leech lattice Λ_{24} is found in 175 seconds.

18.5.1 Example: Automorphism Group of E_8

We compute the automorphism group of the root lattice E_8 and manually transform the action on the coordinates into an action on the lattice vectors.

```
> L := Lattice("E", 8);
> G := AutomorphismGroup(L);
> #G; FactoredOrder(G);
696729600
[ <2, 14>, <3, 5>, <5, 2>, <7, 1> ]
> M := MatrixRing(Rationals(), 8);
> B := BasisMatrix(L);
> A := MatrixGroup<8, Rationals() | [B^-1 * M!G.i * B : i in [1 .. Ngens(G)]]>;
> A;
MatrixGroup(8, Rational Field)
Generators:
    [   0    0 -1/2  1/2 -1/2  1/2    0    0]
    [   0    0  1/2  1/2  1/2  1/2    0    0]
    [   0    0 -1/2  1/2  1/2 -1/2    0    0]
    [-1/2  1/2    0    0    0    0 -1/2  1/2]
    [   0    0 -1/2 -1/2  1/2  1/2    0    0]
    [-1/2 -1/2    0    0    0    0 -1/2 -1/2]
    [-1/2 -1/2    0    0    0    0  1/2  1/2]
    [ 1/2 -1/2    0    0    0    0 -1/2  1/2]


    [ 1/4  1/4  1/4 -1/4 -3/4 -1/4 -1/4 -1/4]
    [-1/4 -1/4  3/4  1/4 -1/4  1/4  1/4  1/4]
    [-1/4 -1/4 -1/4  1/4 -1/4  1/4  1/4 -3/4]
    [-1/4 -1/4 -1/4  1/4 -1/4  1/4 -3/4  1/4]
    [ 1/4 -3/4  1/4 -1/4  1/4 -1/4 -1/4 -1/4]
    [ 3/4 -1/4 -1/4  1/4 -1/4  1/4  1/4  1/4]
    [-1/4 -1/4 -1/4  1/4 -1/4 -3/4  1/4  1/4]
    [-1/4 -1/4 -1/4 -3/4 -1/4  1/4  1/4  1/4]


    [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 0 1]
    [0 0 0 0 0 0 1 0]
> [ #Orbit(A, b) : b in Basis(L) ];
[ 2160, 240, 240, 240, 240, 240, 240, 240 ]
```

Chapter 19

Algebras

19.1 Finite Dimensional Algebras

19.1.1 A Jordan algebra

We define a structure constant algebra which is a Jordan algebra.

```
> M := MatrixAlgebra( GF(3), 2 );
> B := Basis(M);
> C := &cat[Coordinates(M,(B[i]*B[j]+B[j]*B[i])/2) : j in [1..#B], i in [1..#B]];
> A := Algebra< GF(3), #B | C >;
> #A;
81
> IsAssociative(A);
false
> IsLie(A);
false
> IsCommutative(A);
true
```

This is a good start, as one of the defining properties of Jordan algebras is that they are commutative. The other property is that the identity $(x^2 * y) * x = x^2 * (y * x)$ holds for all $x, y \in A$. We check this on a random pair.

```
> x := Random(A); y := Random(A);
> (x^2*y)*x - x^2*(y*x);
(0 0 0 0)
```

The algebra is small enough to check this identity on all elements.

```
> forall{<x, y>: x, y in A | (x^2*y)*x eq x^2*(y*x)};
true
```

So the algebra is in fact a Jordan algebra (which was clear by construction). We finally have a look at the structure constants.

```
> BasisProducts(A);
[
    [ (1 0 0 0), (0 2 0 0), (0 0 2 0), (0 0 0 0) ],
    [ (0 2 0 0), (0 0 0 0), (2 0 0 2), (0 2 0 0) ],
    [ (0 0 2 0), (2 0 0 2), (0 0 0 0), (0 0 2 0) ],
    [ (0 0 0 0), (0 2 0 0), (0 0 2 0), (0 0 0 1) ]
]
```

19.1.2 The real Cayley algebra

We construct the real Cayley algebra, which is a non-associative algebra of dimension 8, containing 7 quaternion algebras. If the basis elements are labelled $1, \ldots, 8$ and 1 corresponds to the identity, these quaternion algebras are spanned by $\{1, (n+1) \bmod 7+2, (n+2) \bmod 7+2, (n+4) \bmod 7+4\}$, where $0 \leq n \leq 6$. We first define a function, which, given three indices i, j, k constructs a sequence with the structure constants for the quaternion algebra spanned by $1, i, j, k$ in the quadruple notation.

```
> quat := func<i,j,k | [<1,1,1, 1>, <i,i,1, -1>, <j,j,1, -1>, <k,k,1, -1>,
>   <1,i,i, 1>, <i,1,i, 1>, <1,j,j, 1>, <j,1,j, 1>, <1,k,k, 1>, <k,1,k, 1>,
>   <i,j,k, 1>, <j,i,k, -1>, <j,k,i, 1>, <k,j,i, -1>, <k,i,j, 1>, <i,k,j, -1>]>;
```

We now define the sequence of non-zero structure constants for the Cayley algebra using the function `quat`. Some structure constants are defined more than once and we have to get rid of these when defining the algebra.

```
> con := &cat[quat((n+1) mod 7 +2, (n+2) mod 7 +2, (n+4) mod 7 +2):n in [0..6]];
> C := Algebra< Rationals(), 8 | Setseq(Set(con)) >;
> C;
Algebra of dimension 8 with base ring Rational Field
> IsAssociative(C);
false
> IsAssociative( sub< C | C.1, C.2, C.3, C.5 > );
true
```

The integral elements in this algebra are those where either all coefficients are integral or exactly 4 coefficients lie in $1/2 + \mathbf{Z}$ in positions i_1, i_2, i_3, i_4, such that i_1, i_2, i_3, i_4 are a basis of one of the 7 quaternion algebras or a complement of such a basis. These elements are called the integral Cayley numbers and form a $\mathbf{Z}$-algebra. The units in this algebra are the elements with either one entry ± 1 and the others 0 or with 4 entries $\pm 1/2$ and 4 entries 0, where the non-zero entries are in the positions as described above. This gives 240 units and they form (after rescaling with $\sqrt{2}$) the roots in the root lattice of type E_8.

```
> a := (C.1 - C.2 + C.3 - C.5) / 2;
> MinimalPolynomial(a);
$.1^2 - $.1 + 1
> MinimalPolynomial(a^-1);
$.1^2 - $.1 + 1
> MinimalPolynomial(C.2+C.3);
$.1^2 + 2
> MinimalPolynomial((C.2+C.3)^-1);
```

```
$.1^2 + 1/2
```

Tensoring the integral Cayley algebra with a finite field gives a finite Cayley algebra. As the **Z**-algebra generated by the chosen basis for C has index 2^4 in the full integral Cayley algebra, we can get the finite Cayley algebras by applying the `ChangeRing` function for finite fields of odd characteristic. The Cayley algebra over $GF(q)$ has the simple group $G_2(q)$ as its automorphism group. Since the identity has to be fixed, every automorphism is determined by its image on the remaining 7 basis elements. Each of these has minimal polynomial $x^2 + 1$, hence one obtains a permutation representation of $G_2(q)$ on the elements with this minimal polynomial. As $\pm$-pairs have to be preserved, this number can be divided by 2.

```
> C3 := ChangeRing( C, GF(3) );
> f := MinimalPolynomial(C3.2);
> f;
$.1^2 + 1
> #C3;
6561
> time Im := [ c : c in C3 | MinimalPolynomial(c) eq f ];
Time: 3.099
> #Im;
702
> C5 := ChangeRing( C, GF(5) );
> f := MinimalPolynomial(C5.2);
> f;
$.1^2 + 1
> #C5;
390625
> time Im := [ c : c in C5 | MinimalPolynomial(c) eq f ];
Time: 238.620
> #Im;
15750
```

In the case of the Cayley algebra over $GF(3)$ we obtain a permutation representation of degree 351, which is in fact the smallest possible degree (corresponding to the representation on the cosets of the largest maximal subgroup $U_3(3) : 2$). Over $GF(5)$, the permutation representation is of degree 7875, corresponding to the maximal subgroup $L_3(5) : 2$, the smallest possible degree being 3906.

19.2 Group Algebras

19.2.1 Diameter of the Cayley graph

We use the group algebra to determine the diameter of the Cayley graph of a group.

```
> G := Alt(6);
> QG := GroupAlgebra( Rationals(), G );
> e := QG!1 + &+[ QG!g : g in Generators(G) ];
> e;
Id(G) + (1, 2)(3, 4, 5, 6) + (1, 2, 3)
```

The group elements that can be expressed as words of length at most n in the generators of G have non-zero coefficient in e^n. The following function returns for a group algebra element e a sequence with the cardinalities of the supports of e^n and breaks when the group order is reached.

```
> wordcount := function(e)
>     f := e;
>     count := [ #Support(f) ];
>     while count[#count] lt #Group(Parent(e)) do
>         f *:= e;
>         Append(~count, #Support(f));
>     end while;
>     return count;
> end function;
```

Now apply this function to the above defined element:

```
> wordcount( e );
[ 3, 7, 14, 26, 47, 83, 140, 219, 293, 345, 360 ]
```

Thus, every element in A_6 can be expressed as a word of length at most 11 in the generators $(1,2)(3,4,5,6)$ and $(1,2,3)$. A better 2-generator set is for example $(1,2,3,4,5)$ and $(1,5,3,6,4)$, where all elements can be expressed as words of length at most 10 and this is in fact optimal. A worst 2-generator set is given by $(1,2)(3,4)$ and $(1,5,3,2)(4,6)$.

```
> wordcount( QG!1 + G!(1,2,3,4,5) + G!(1,5,3,6,4) );
[ 3, 7, 15, 31, 60, 109, 183, 274, 350, 360 ]
> wordcount( QG!1 + G!(1,2)(3,4) + G!(1,5,3,2)(4,6) );
[ 3, 6, 11, 18, 28, 43, 63, 88, 119, 158, 206, 255, 297, 329, 352, 360 ]
```

19.2.2 Random distribution of words

The group algebra can also be used to investigate the random distribution of words of a certain length in the generators of the group.

```
> M11 := sub< Sym(11) | (1,11,9,10,4,3,7,2,6,5,8), (1,5,6,3,4,2,7,11,9,10,8) >;
> A := GroupAlgebra(RealField(16), M11 : Rep := "Vector");
> A;
Group algebra with vector representation
Coefficient ring: Real Field of precision 16
Group: Permutation group M11 acting on a set of cardinality 11
    Order = 7920 = 2^4 * 3^2 * 5 * 11
        (1, 11, 9, 10, 4, 3, 7, 2, 6, 5, 8)
        (1, 5, 6, 3, 4, 2, 7, 11, 9, 10, 8)
> e := (A!M11.1 + A!M11.2) / 2.0;
> eta := Eta(A) / #M11;
```

For growing n, the words of length n in the generators of `M11` converge towards a random distribution iff $\texttt{e}^n$ converges towards `eta`. We look at the quadratic differences of the coefficients of $\texttt{e}^n - \texttt{eta}$ for $n = 10, 20, 30, 40, 50$.

```
> e10 := e^10;
> f := A!1;
> for i in [1..5] do
>     f *:= e10;
>     print &+[ c^2 : c in Eltseq(f - eta) ];
> end for;
0.0012050667195213
1.289719354694155e-5
5.9390965208879e-7
3.394099291966e-8
2.19432454574986e-9
```

19.3 Matrix Algebras

19.3.1 Jordan forms of matrices over a finite field

Set A to be a 28 by 28 permutation matrix with entries in finite field with 3 elements.

```
> A := ActionGenerator(GModule(PSU(3, 3), GF(3)), 1);
> A;
[1 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 1 0 0 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 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 1 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 1 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 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 1 0 0 0 0]
[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 1 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 1 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 0 0 1]
[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 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 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 0 1 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 1 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 1 0 0 0 0 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 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 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 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 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 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 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 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 1 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 1 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 1 0 0]
```

Calculate the characteristic polynomial c of A and factorize c—note that there are quadratic irreducible factors.

```
> c<x> := CharacteristicPolynomial(A);
> c;
x^28 + x^27 + 2*x^25 + 2*x^24 + 2*x^4 + 2*x^3 + x + 1
> Factorization(c);
[
    <x + 1, 4>,
    <x + 2, 6>,
    <x^2 + 1, 3>,
    <x^2 + x + 2, 3>,
    <x^2 + 2*x + 2, 3>
]
```

Calculate the primary invariant factors of A which give the structure of the blocks in the (generalized) Jordan form of A.

```
> PrimaryInvariantFactors(A);
[
    <x + 1, 1>,
    <x + 1, 1>,
    <x + 1, 1>,
    <x + 1, 1>,
    <x + 2, 1>,
    <x + 2, 1>,
    <x + 2, 1>,
    <x + 2, 1>,
    <x + 2, 1>,
    <x + 2, 1>,
    <x^2 + 1, 1>,
    <x^2 + 1, 1>,
    <x^2 + 1, 1>,
    <x^2 + x + 2, 1>,
    <x^2 + x + 2, 1>,
    <x^2 + x + 2, 1>,
    <x^2 + 2*x + 2, 1>,
    <x^2 + 2*x + 2, 1>,
    <x^2 + 2*x + 2, 1>
]
```

To get all the factors to split, we set F to be a degree 2 extension of F.

```
> F<w> := ext< GF(3) | 2 >;
```

Now we let AF be the matrix A but with entries in F and then let cF be the characteristic polynomial of AF. Note that cF now splits completely.

```
> AF := MatrixAlgebra(F, 28) ! A;
> cF<y> := CharacteristicPolynomial(AF);
```

```
> cF;
y^28 + y^27 + 2*y^25 + 2*y^24 + 2*y^4 + 2*y^3 + y + 1
> Factorization(cF);
[
    <y + 1, 4>,
    <y + w, 3>,
    <y + w^2, 3>,
    <y + w^3, 3>,
    <y + 2, 6>,
    <y + w^5, 3>,
    <y + w^6, 3>,
    <y + w^7, 3>
]
```

Finally, AF possesses a true Jordan form over F (in fact, it is diagonalizable).

```
> JordanForm(AF);
[2 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 2 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 2 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 2 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 w^5 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 w^5 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 w^5 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 w^6 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 w^6 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 w^6 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 w^7 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 w^7 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 w^7 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 1 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 1 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 1 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 1 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 1 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 1 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 w 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 w 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 w 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 w^2 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 w^2 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 w^2 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 w^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 w^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 w^3]
```

19.3.2 Jordan forms of matrices over the rationals

Set A to be a matrix with entries in the rational field.

```
> M := MatrixRing(RationalField(), 10);
```

```
> A := M ! [i * j : i, j in [1 .. 10]];
> A;
[  1   2   3   4   5   6   7   8   9  10]
[  2   4   6   8  10  12  14  16  18  20]
[  3   6   9  12  15  18  21  24  27  30]
[  4   8  12  16  20  24  28  32  36  40]
[  5  10  15  20  25  30  35  40  45  50]
[  6  12  18  24  30  36  42  48  54  60]
[  7  14  21  28  35  42  49  56  63  70]
[  8  16  24  32  40  48  56  64  72  80]
[  9  18  27  36  45  54  63  72  81  90]
[ 10  20  30  40  50  60  70  80  90 100]
```

Note the characteristic polynomial and minimal polynomial of A.

```
> c<x> := CharacteristicPolynomial(A);
> c;
x^10 - 385*x^9
> Factorization(c);
[
    <x, 9>,
    <x - 385, 1>
]
> MinimalPolynomial(A);
x^2 - 385*x
```

Find the Jordan form J, with the transformation matrix T, and the primary invariant factors F.

```
> J, T, F := JordanForm(A);
> J;
[385   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   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   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0]
> T;
[     1      2      3      4      5      6      7      8      9     10]
[     1      2      3      4      5      6      7      8      9  -57/2]
[     1      2      3      4      5      6      7      8 -304/9     10]
[     1      2      3      4      5      6      7 -321/8      9     10]
[     1      2      3      4      5      6    -48      8      9     10]
[     1      2      3      4      5 -349/6      7      8      9     10]
[     1      2      3      4    -72      6      7      8      9     10]
[     1      2      3 -369/4      5      6      7      8      9     10]
[     1      2 -376/3      4      5      6      7      8      9     10]
[  -384      2      3      4      5      6      7      8      9     10]
> F;
```

```
[
    <x - 385, 1>,
    <x, 1>,
    <x, 1>,
    <x, 1>,
    <x, 1>,
    <x, 1>,
    <x, 1>,
    <x, 1>,
    <x, 1>,
    <x, 1>
]
```

Check that T conjugates A to J.

```
> J eq T * A * T^-1;
true
```

19.3.3 Matrix algebra over a polynomial ring

We consider the algebra $M_5(P)$, where P is the polynomial ring in indeterminate x over the field GF(5). We take the matrix having $x^i + x^j$ in its (i, j)-th position.

```
> K := GaloisField(5);
> P<x> := PolynomialAlgebra(K);
> M := MatrixAlgebra(P, 5);
> a := M ! [x^i + x^j: i, j in [1..5]];
> a;
[      2*x   x^2 + x   x^3 + x   x^4 + x   x^5 + x]
[  x^2 + x     2*x^2 x^3 + x^2 x^4 + x^2 x^5 + x^2]
[  x^3 + x x^3 + x^2     2*x^3 x^4 + x^3 x^5 + x^3]
[  x^4 + x x^4 + x^2 x^4 + x^3     2*x^4 x^5 + x^4]
[  x^5 + x x^5 + x^2 x^5 + x^3 x^5 + x^4     2*x^5]
> ElementaryDivisors(a);
[
    x,
    x^3 + 3*x^2 + x
]
```

19.3.4 Orders of a unit in a matrix ring

We calculate the order and factored order of a random matrix over a finite field.

```
> M := MatrixAlgebra(GF(23), 20);
> A := Random(M);
> A;
[ 6  1 14  6 11 21 16  7  7 15  3 21  7  5 21 17  7  6 16  1]
[15 17  5  4 13  5 18 21 11 12 18  1  7 22 12  1 19 22  9 15]
```

```
[21 13  4 13 17  0  0 21  3  0 13 13  0  1 17  9  9 18 11 21]
[14  5 20 20 22  4  5 21  0  3  7 14  1  0 12 19  5 18  0 20]
[19  9 14 19  4  5 20  8 22  8 12  9  9 22  9 16 10 14  8  5]
[21 11 12  6 11 19 12  9  8  9  7  0 10  0  3 16 21  2 19  9]
[14  9  9 16 22  5  0 14  6  4  2 11 20 17  7 10  7  7 13 10]
[22 21  2 22 11 18  7  3 19  7  2 18 11  3 10 18 10  8  1 19]
[ 5 17 10 17  1 22  8  3 19 13 22 20  8 12 17 14  3 15 12  4]
[ 0 16  6  7 19 19 10  3 15 21  3 22 13 19 22  6 19  1 12 12]
[18 19 18  0 15  5 19 22  6  9 22 20 16 17 12  2  5  2 22 16]
[21  1 22  6 18 14  2  7  8 15  9 20 11 15 20  7 16  3  5  8]
[ 8 19 18  3  7  5  7 19 22 13  4 13  7  4 11 21  3 14  8  3]
[ 5 11 15 15 19  0  1 12  0  8  0  1 18 10  8  0  5  0 15 11]
[21  7 18  2  5 22 21  8  6  5 18 17 22 15 12 13  2  7  6  4]
[ 3  7 13 12 19  3 10 16 18 20 10 21 11 21  2 19 11  6 13  8]
[15 20 18 15 12  7 18  2  3 16 18  4  7 14 17 16  0 22 15  1]
[15 11 21 12  9  2  0 12 12 21 12 10 11 20  8  2 10 17 13 21]
[15 10 17  0 19 13 21 21 16 10 13  0 10 12 13 21  3 10 20  7]
[ 3 21 11 12 16 13  2 17 21 12 16 11 14  9  7 10 19 10  0  7]
>
> Order(A);
216138319375440
> FactoredOrder(A);
[ <2, 4>, <3, 1>, <5, 1>, <7, 1>, <11, 1>, <19, 1>, <79, 1>, <7792003, 1> ]
```

19.4 Lie Algebras

We create the Lie algebra $sl_3(\mathbb{Q})$ as a structure constant algebra. First, we construct $gl_3(\mathbb{Q})$ from the full matrix algebra $M_3(\mathbb{Q})$ and get $sl_3(\mathbb{Q})$ as the derived algebra of $gl_3(\mathbb{Q})$.

```
> gl3 := LieAlgebra(Algebra(MatrixRing(Rationals(), 3)));
> sl3 := gl3 * gl3;
> sl3;
Lie Algebra of dimension 8 with base ring Rational Field
```

Let's see how the first basis element acts.

```
> for i in [1..8] do
>     print sl3.i * sl3.1;
> end for;
(0 0 0 0 0 0 0 0)
( 0 -1  0  0  0  0  0  0)
( 0  0 -2  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 2 0)
(0 0 0 0 0 0 0 1)
```

Since it acts diagonally, this element lies in a Cartan subalgebra. The next candidate seems to be the fifth basis element.

```
> for i in [1..8] do
>     print sl3.i * sl3.5;
> end for;
(0 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 0 0 0 0)
( 0  0  0  0  0 -2  0  0)
(0 0 0 0 0 0 1 0)
(0 0 0 0 0 0 0 2)
```

This also acts diagonally and commutes with `sl3.1`, hence we have luckily found a full Cartan algebra in $sl_3(\mathbb{Q})$. We can now easily work out the root system. Obviously the root spaces correspond to the pairs `(sl3.2, sl3.4)`, `(sl3.3, sl3.7)` and `(sl3.6, sl3.8)`. The product of a positive root with its negative should lie in the Cartan algebra.

```
> sl3.2*sl3.4;
( 1  0  0  0 -1  0  0  0)
> sl3.3*sl3.7;
(1 0 0 0 0 0 0 0)
> sl3.6*sl3.8;
(0 0 0 0 1 0 0 0)
```

Clearly some choices have to be made and we fix `sl3.3` as the element e_α corresponding to the first fundamental root α, `sl3.7` as $e_{-\alpha}$ and get `sl3.1` as $h_\alpha = e_\alpha * e_{-\alpha}$. For the other fundamental root β we have to find an element e_β such that $e_\alpha * e_\beta$ is non-zero.

```
> sl3.3*sl3.2;
(0 0 0 0 0 0 0 0)
> sl3.3*sl3.4;
( 0  0  0  0  0 -1  0  0)
> sl3.3*sl3.6;
(0 0 0 0 0 0 0 0)
> sl3.3*sl3.8;
(0 1 0 0 0 0 0 0)
```

We choose `sl3.8` as e_β, `sl3.6` as $e_{-\beta}$ and consequently `-sl3.5` as h_β. This now determines $e_{\alpha+\beta}$ to be `sl3.2` and $e_{-\alpha-\beta}$ to be `sl3.4`.

19.5 Finitely presented algebras

19.5.1 Hecke algebra

Each Coxeter group has associated with it a Hecke algebra generated by elements corresponding to a set of fundamental roots for the group. Although the Hecke algebra is normally defined over the ring of Laurent polynomials $\mathbf{Z}[q, q^{-1}]$, we may construct a presentation by extending to $\mathbb{Q}$ and setting q equal to a primitive root of unity. The following Magma code constructs a presentation

for the Hecke algebra of type E_6. Linton's vector enumerator[1]is then applied to construct a 27-dimensional representation over the field $\mathbb{Q}(\omega)$, where ω is a fifth root of unity.

```
> fhm<x,y,z,t,u,v,q>  := FreeMonoid(7);
> hm := quo<fhm| x*y*x = y*x*y, z*y*z = y*z*y, x*z = z*x, x*t = t*x, y*t = t*y,
>       t*z*t = z*t*z, u*x = x*u, u*y = y*u, u*z*u = z*u*z, u*t = t*u,
>       x*v = v*x, y*v = v*y, z*v = v*z, t*v = v*t, u*v*u = v*u*v>;
>
> fha<x, y, z, t, u, v, q> := FreeAlgebra(CyclotomicField(5), hm);
>
> ha<x, y, z, t, u >, ham := quo<fha |
>                                 x*x = (q-1)*x+q,
>                                 y*y = (q-1)*y+q,
>                                 z*z = (q-1)*z+q,
>                                 t*t = (q-1)*t+q,
>                                 u*u = (q-1)*u+q,
>                                 v*v = (q-1)*v+q,
>                                 q = fha!RootOfUnity(5, CyclotomicField(5)) >;
> hi := rideal<ha | x+1,y+1,z+1,t+1,u+1>;
> Q, im, f := QuotientModule(ha, hi);
> Degree(im[1]);
27
```

[1]S.A. Linton, "Generalizations of the Todd-Coxeter Algorithm", in: Wieb Bosma, Alf van der Poorten (eds), *Computational Algebra and Number Theory*, Sydney 1992, Dordrecht: Kluwer, to appear, 1995; and S.A. Linton, "Constructing Matrix Representations of Finitely Presented Groups", *J. Symbolic Comput.* **12** (1991), 427–438.

Chapter 20

Plane Curves

The features to be released in MAGMA version 2.5 are:

- Affine and projective space and their points
- Affine and projective curves and schemes
- Basic analysis of points on schemes
- Maps between spaces
- Linear systems on projective spaces
- Newton Polygons
- Divisor groups on certain curves

20.1 Affine curve singularities

Curves in the plane are defined as the vanishing of a polynomial, $y - x^2 = 0$ defining a parabola for instance. But before making a curve in MAGMA, one must make the plane, its ambient space. I make an affine plane over the rationals with coordinates x, y as follows.

```
> k := Rationals();
> A<x,y> := AffineSpace(k,2);
> A;
Affine space of dimension 2 with coordinates x,y
```

Of course, one can create other affine spaces: maybe the intersection points of two curves will have coordinates in some extension of the rationals and a new plane with these extended coefficients will be needed to find them.

One defines points in affine spaces using MAGMA's usual coercion operator `!` to reinterpret sequences of coordinates.

```
> p := A ! [1,0];
> p;
[ 1, 0 ]
```

The ambient space records the base field over which all points, curves and other schemes lying in it must be defined. So the point $[\sqrt{2}, 0]$ could not have been coerced into this ambient space.

One creates curves using a reference to a 2-dimensional ambient space and a polynomial defined on it.

```
> C := Curve(A, x^2 + y^3);
> C;
Affine plane curve defined by x^2 + y^3
```

This curve is contained in A and will stay that way: if you want to change it, by a base extension or change of coordinates say, you will have to create a new curve.

```
> Ambient(C) eq A;
true
> BaseRing(C);
Rational Field
```

There are a number of functions to look for singularities of a plane curve.

```
> IsNonSingular(C);
false
> SingularPoints(C);
[
    [0, 0]
]
> q := SingularPoints(C)[1];
> T := TangentCone(C,q);
> T;
Affine scheme defined by [ x^2 ]
> IsReduced(T);
false
```

Notice that the tangent cone is embedded as a scheme in the same ambient space as C itself. The code above could be the basis of a routine to check whether a given plane curve has only ordinary singularities.

The function `Blowup()` returns the two standard affine plane curve patches on the blowup at the origin.

```
> Blowup(C);
Affine plane curve defined by x*y^3 + 1
Affine plane curve defined by x^2 + y
```

In this case a single blowup resolves C — indeed, I can see that both patches are nonsingular. For more complicated singularities more blowups will be needed. In a later version, MAGMA will include functions to carry out the complete resolution of C expressing the data in a variety of ways.

The curve C is clearly of genus zero:

```
> Genus(C);
0
```

Since its equation has only two terms one can easily spot a parametrisation.

```
> A1<t> := AffineSpace(k,1);
> f := RationalMap(A1,A,[t^3,-t^2]);
> Image(f) eq C;
true
```

20.2 A canonical embedding

The syntax of projective space is very similar to that of affine space.

```
> P<x,y,z> := ProjectiveSpace(k,2);
> f := x^5 + y^4*z + y^2*z^3;
> C := Curve(P,f);
> C;
Projective plane curve defined by x^5 + y^4*z + y^2*z^3
> SingularPoints(C);
[
    [ 0, 0, 1 ]
]
> Genus(C);
4
```

Maps from one projective space to another are defined by a sequence of homogeneous polynomials of equal degree. I choose such a sequence out of the blue; later on I give some justification for my choice.

```
> P3<a,b,c,d> := ProjectiveSpace(k,3);
> phi := RationalMap(P,P3,[x^2,x*y,y^2,y*z]);
> phi;
Map of projective spaces defined by [ x^2, x*y, y^2, y*z ]
> Image(phi);
Projective scheme defined by [ -a*c + b^2 ]
```

The map ϕ is a birational embedding of the projective plane into projective 3-space. Its image, strictly speaking, the closure of the image of the regular part of ϕ, is the singular quadric cone $b^2 = ac$.

I really want to calculate the image of C under this map. That is harder to do, but I can at least calculate the equations of the image in any chosen degree. Of course, these might not be enough to cut out C precisely, but a finite number of such equations will be.

```
> IC2 := Image(phi,C,2);
> IC2;
[ a*c - b^2 ]
```

That's good — we certainly expect C to lie in the quadric cone since that's the image of the plane which C started out in — but we clearly need to find more equations.

```
> IC3 := Image(phi,C,3);
> X := Scheme(P3,IC2 cat IC3);
> Dimension(X);
1
> IsNonSingular(X);
true
> MinimalBasis(X);
[ a*c - b^2, a^2*b + c^2*d + d^3 ]
```

So I have expressed the image of C as the nonsingular intersection of a conic and a cubic in projective 3-space; compare with Hartshorne, *Algebraic Geometry*, GTM 52, Springer (1977), section IV example (5.2.2) where one sees that any nonhyperelliptic curve of genus 4 admits such a description.

To understand the choice of map in the previous paragraph, I analyse C by blowing up its singular point a couple of times. I use some knowledge of curve singularities and their adjoints.

```
> C1<u,v> := AffinePatch(C,1);
> C1;
Affine plane curve defined by u^5 + v^4 + v^2
> Blowup(C1);
Affine plane curve defined by u^3 + u^2*v^4 + v^2
Affine plane curve defined by u^5*v^3 + v^2 + 1
> C2 := Blowup(C1);
> Blowup(C2);
Affine plane curve defined by u^4*v^4 + u + v^2
Affine plane curve defined by u^3*v + u^2*v^4 + 1
```

So I see that C has a single double point at $(0:0:1)$. Blowing that up produces another double point, a cusp, which is resolved by one more blowup. The expected genus of a degree 5 plane curve is $\frac{1}{2}(5-1)(5-2) = 6$. Standard results in the resolution of curve singularities show that the genus of this curve is $6-1-1=4$ as was calculated earlier. Moreover, the (canonical) adjoints to C are those plane curves of degree 2 passing through both the double point and the double point after the blowup. I calculate these using linear systems.

Given the system of all curves of degree 2, I first restrict to those curves passing through the double point, and then to those having a specified tangent at the double point, a condition equivalent to the blowup condition.

```
> L := LinearSystem(P,2);
> L;
The complete linear system of degree 2
> p := SingularPoints(C)[1];
> L1 := Subsystem(L,p);
> L1;
The linear system defined by [ x^2, x*y, x*z, y^2, y*z ]
```

The system L_1 is clearly too big: I'm essentially calculating a basis of sections of the canonical class of the resolution of C, and the genus is by definition the size of such a basis.

```
> T := ProjectivePlaneCurveType(Reduced(TangentCone(C,p)));
> L2 := Subsystem(L,p,T);
> L2;
The linear system defined by [ x^2, x*y, y^2, y*z ]
```

It is these functions that I used to define a map from the projective plane to projective 3-space above.

20.3 Birational maps of the projective plane

Automorphisms of the projective plane are always linear. However, there are lots of maps which are not everywhere defined, but which are bijections on an open subset of the plane. These are the so-called *birational automorphisms*, often called *cremona transformations*. The archetype is the following, called the *standard quadratic transformation*.

```
> P<x,y,z> := ProjectiveSpace(k,2);
> f := RationalMap(P,P,[1/x,1/y,1/z]);
> f;
Map of projective spaces defined by [ y*z, x*z, x*y ]
```

Notice that the map is normalised by multiplying up the denominators. I say that f has *function degree* 2 since it can be defined using quadratic polynomials with no common factors. A cremona transformation of the plane is a genuine automorphism if and only if it has function degree 1.

At the end of the last century, Max Nöther and others showed that any cremona transformation can be composed (up to linear automorphisms) with the standard quadratic transformation to produce a cremona transformation of lower function degree. (In fact, there are cases when this doesn't work, but after a sequence of such compositions it always does.) By making a string of such compositions, he showed that any cremona transformation factorises as a composition of linear automorphisms and standard quadratic transformations. Consider an example.

```
> funs := [2/3*x^2*y^2 + 2/3*x^2*y*z + x*y^2*z + x*y*z^2,
           1/3*x^2*y^2 + 4/3*x^2*y*z + x^2*z^2 + 1/2*x*y^2*z + 1/2*x*y*z^2,
           2/9*x^2*y^2 + 2/3*x^2*y*z + 2/3*x*y^2*z + x*y*z^2 + 1/2*y^2*z^2 ];
> g := RationalMap(P,P,funs);
> FunctionDegree(g);
4
```

How do you begin? The trick is to find three points in the plane where g is not defined, move them to the standard coordinate points and make a quadratic transformation. (The existence of such points in a generalised setting and then the correct choice of a subset of them is determined by the Nöther–Fano inequalities. However, I don't do that here, rather I get ignorantly lucky.)

```
> Support(BaseLocus(g));
[ [ 0, 0, 1 ], [ -3/2, -1, 1 ], [ 3/4, -1, 1 ], [ 0, 1, 0 ], [ 1, 0, 0 ] ]
> std_quad := QuadraticTransformation(P);
> g1 := Composition(g,std_quad);
> FunctionDegree(g1);
2
```

The function degree has dropped which is the thing directing the induction. So I continue.

```
> Support(BaseLocus(g1));
[ [ -2/3, 0, 1 ], [ -2/3, -1, 1 ], [ 4/3, -1, 1 ] ]
> tr := Translation(P,$1);
> quad := Composition(tr,std_quad);
> g2 := Composition(g1,quad);
> FunctionDegree(g2);
1
```

So I'm done: the function g_2 is a linear automorphism. To check the factorisation I compose the maps appearing in the algorithm in the reverse order and recover g; note that standard quadratic transformations are selfinverse.

```
> f1 := Composition(Inverse(tr),std_quad);
> f2 := Composition(std_quad,f1);
> f3 := Composition(g2,f2);
> f3 eq g;
true
```

20.4 Linear equivalence of divisors

I don't explain the theory of divisors here except to say that one can think of them as being formal sums of points of a curve; in particular, they form a group.

```
> P<x,y,z> := ProjectiveSpace(FiniteField(5),2);
> E := Curve(P,y^2*z-x^3-2*x*z^2 - z^3);
> D := DivisorGroup(E);
> D;
Divisor group of the Projective plane curve defined by 4*x^3 +
    3*x*z^2 + y^2*z + 4*z^3
```

Notice that in this example the curve E is elliptic and in Weierstraß form. This is currently a requirement of the functions below.

Given a point on E, one considers it as a divisor by coercing it into the divisor group.

```
> p := P ! [3,2,1];
> d := D ! p;
> d;
[ < [ 3, 2, 1 ], 1 > ]
```

The trailing 1 is the coefficient of the formal sum $1 \cdot p$. A line in the plane can be used to create a principal divisor, the divisor of zeros and poles of the rational function ℓ/z where ℓ is the equation of the line.

```
> L := Divisor(D,p,p);
> L;
[ < [ 3, 2, 1 ], 2 >, < [ 0, 4, 1 ], 1 >, < [ 0, 1, 0 ], -3 > ]
```

Notice the coefficient -3 coming from the pole of the flex $z = 0$ to E at infinity.

In the case of genus 1 curves like E one can always add a number of divisors of this form to a given divisor to reduce it to a normal form having only a single positive component other than the point at infinity $(0 : 1 : 0)$. See what I mean in an example: the normal form is mostly supported on the point $(0 : 1 : 0)$.

```
> 3*d;
[ < [ 3, 2, 1 ], 3 > ]
> NormalForm(3*d);
[ < [ 1, 2, 1 ], 1 >, < [ 0, 1, 0 ], 2 > ]
```

Two divisors are said to be *linearly equivalent* if their difference is the divisor of zeros and poles of some rational function, or equivalently, if their normal forms are identical.

```
> AreLinearlyEquivalent(3*d,$1);
true
($.1^2 + 4*$.1*$.2 + $.1*$.3 + 3*$.2*$.3 + 3*$.3^2)/
    ($.1*$.3 + 3*$.2*$.3 + 3*$.3^2)
```

Notice that a rational function is also returned. (Names for the generators of the rational functions are not known, so the `$.i` notation is used.) This rational function corresponds to the difference of the two arguments. It is unique up to a scalar multiple. The algorithm to calculate it is straightforward: choose particular divisors arising from lines to add to or subtract from the difference of the arguments — simply aim to eliminate points not at infinity by cancellation; each of these generates a quotient of linear polynomials whose product is the required function.

Chapter 21

Elliptic Curves

21.1 Elliptic Curves over a General Field

Elliptic curves may be created over any field supported in MAGMA. The use of generic code enables basic arithmetic of points to be performed regardless of the base field. The following general operations are available:

- Creation of an elliptic curve over a field
- Creation of a curve with given j-invariant
- Invariants: b-invariants, c-invariants, j-invariant, discriminant
- Arithmetic with rational points
- Extension and lifting of curves induced by maps of base rings
- Division polynomials

21.1.1 Example: Generic point on an elliptic curve

We construct a generic point on an elliptic curve, by base extending the curve to its function field.

First we construct the curve:

```
> E := EllipticCurve([GF(97) | 1, 2]);
```

Next, we form its function field;

```
> a1, a2, a3, a4, a6 := Explode(aInvariants(E));
> R := BaseRing(E);
> Px<x> := FunctionField(R);
> Py<Y> := PolynomialRing(Px);
> F<y> := quo< Py | (Y^2 + (a1*x + a3)*Y - (x^3 + a2*x^2 + a4*x + a6))>;
```

and change the base ring of the curve to be that field.

```
> EF := BaseChange(E, F);
```

Now we can construct a generic point, and show that doubling the point results in the usual expression for the double of a point.

```
> gen_pt := EF![x,y];
> gen_pt;
(x, y, 1)
> 2*gen_pt;
((73*x^4 + 48*x^2 + 93*x + 73)/(x^3 + x + 2), (85*x^6 + 37*x^4 + 5*x^3
    + 60*x^2 + 96*x + 8)/(x^6 + 2*x^4 + 4*x^3 + x^2 + 4*x + 4)*y, 1)
```

The formulas for multiplication by n, and even for the addition law can be "discovered" in this way.

21.2 Subgroups and Subschemes of Elliptic Curves

MAGMA allows the formation of subgroups and subschemes of elliptic curves. Subgroups are defined by a univariate polynomial; the rational points of the curve which are roots of this polynomial are exactly the x-coordinates of points in the subgroup. General subschemes need not be closed under the group law, and are specified by an ideal of a bivariate polynomial ring. Subgroups arise naturally as the kernels of isogenies as covered in the next section.

- Subgroups and subschemes of elliptic curves as distinct types;
- Make an arbitrary subgroup or subscheme defined by a polynomial or ideal entered by the user;
- Construction of the kernel of a given isogeny as a subgroup;
- Forming the subgroup of all m-torsion points, $m = 1, 2, \ldots$;
- Image of a subgroup under an isogeny.

21.3 Maps Between Elliptic Curves

One of the unique features of MAGMA is its facility for constructing maps between elliptic curves. Four types of maps are supported, isogenies, isomorphisms, translations, and rational maps. Features include:

- Isomorphisms, isogenies and rational maps between curves, translation maps on a curve
- Arithmetic of isomorphisms and isogenies: inverses, composition
- Degree of an isogeny;
- Kernel of an isogeny as a subgroup;
- Decide if curves are isomorphic;

- Given a subgroup of an elliptic curve, construction of a separable isogeny with kernel that subgroup;
- Construction of multiplication endomorphisms; given a curve E, isogenies $I_t : E \to E$ such that $I_t(P) = t * P$, for $t = 1, 2, \ldots$;
- Formation of the Frobenius isogeny for curves over finite fields;

21.3.1 Example: Generic isogeny of an elliptic curve

We show to to construct an isogeny with given kernel, and then calculate its dual.

First we construct a separable isogeny whose kernel is the set of 3-torsion points on the curve E.

```
> E := EllipticCurve([GF(97) | 2, 3]);
> E1, I := IsogenyFromKernel(E, DivisionPolynomial(E, 3));
> I;
Elliptic curve isogeny from: CurveEll: E to CurveEll: E1
taking (x, y, 1) to (x^9 + 73*x^7 + 3*x^6 + 23*x^5 + 50*x^4 + 41*x^3 + 91*x^2 +
    29*x + 77 / x^8 + 8*x^6 + 24*x^5 + 78*x^4 + 96*x^3 + 4*x^2 + 65*x + 88,
    $.1^12*$.2 + 44*$.1^10*$.2 + 78*$.1^9*$.2 + 19*$.1^8*$.2 + 33*$.1^7*$.2 +
    61*$.1^6*$.2 + 64*$.1^5*$.2 + 33*$.1^4*$.2 + 13*$.1^3*$.2 + 70*$.1^2*$.2 +
    85*$.1*$.2 + 21*$.2 / $.1^12 + 12*$.1^10 + 36*$.1^9 + 44*$.1^8 + 94*$.1^7 +
    76*$.1^6 + 92*$.1^5 + 85*$.1^4 + 83*$.1^3 + 92*$.1^2 + 64*$.1 + 12, 1)
```

We know the composite of the isogeny and its dual will have kernel the group of n-torsion points, where n is the degree of I. We construct this subgroup and push it through the isogeny:

```
> deg := Degree(I);
> deg;
9
> S := nTorsionSubgroup(E, deg);
> S;
Subgroup of E defined by 9*x^40 + 51*x^38 + 87*x^37 + 4*x^36 + 70*x^35 + 82*x^34
    + 14*x^33 + 67*x^32 + 9*x^31 + 4*x^30 + 17*x^29 + 71*x^28 + 21*x^27 +
    13*x^26 + 86*x^25 + 37*x^24 + 52*x^23 + 64*x^22 + 93*x^21 + 60*x^20 +
    11*x^19 + 73*x^18 + 21*x^17 + 31*x^16 + 7*x^15 + 60*x^14 + 40*x^13 + 63*x^12
    + 90*x^11 + 20*x^10 + 29*x^9 + 4*x^8 + 27*x^7 + 50*x^6 + 3*x^5 + 9*x^4 +
    39*x^3 + 75*x^2 + 65*x + 27
> S1 := PushThroughIsogeny(I, S);
> S1;
Subgroup of E1 defined by x^4 + 33*x^2 + 18*x + 79
```

Next we make the isogeny with kernel np;

```
> E2, J := IsogenyFromKernel(S1);
> E2;
Elliptic Curve defined by y^2 = x^3 + 59*x + 52 over GF(97)
```

Note that $E2$ is not equal to E - they are different models of the same curve. We have to multiply J by the isomorphism between $E2$ and E to get the dual of I.

```
> bool, iso := IsIsomorphic(E2, E);
> error if not bool, "ERROR: curves are not isomorphic, something's wrong";
> iso;
Elliptic curve isomorphism from: CurveEll: E2 to CurveEll: E
Taking (x, y, 1) to (6*x, 33*y, 1)
> Idual := J*iso;
> Idual;
Elliptic curve isogeny from: CurveEll: E1 to CurveEll: E
taking (x, y, 1) to (6*x^9 + 73*x^7 + 27*x^6 + 20*x^5 + 75*x^4 + 20*x^3 + 47*x^2
    + 41*x + 80 / x^8 + 66*x^6 + 36*x^5 + 83*x^4 + 24*x^3 + 9*x^2 + 31*x + 33,
33*$.1^12*$.2 + 48*$.1^10*$.2 + 78*$.1^9*$.2 + 74*$.1^8*$.2 + 54*$.1^7*$.2 +
    43*$.1^6*$.2 + 88*$.1^5*$.2 + 81*$.1^4*$.2 + 5*$.1^3*$.2 + 11*$.1^2*$.2 +
    44*$.1*$.2 + 83*$.2 / $.1^12 + 2*$.1^10 + 54*$.1^9 + 12*$.1^8 + 72*$.1^7 +
    74*$.1^6 + 20*$.1^5 + 44*$.1^4 + 74*$.1^3 + 30*$.1^2 + 36*$.1 + 85, 1)
```

And check:

```
> pt := Random(E);
> Idual(I(pt)) eq deg*pt;
true
```

21.4 Elliptic Curves over the Rational Numbers

Magma contains an extensive package of functions for computing with elliptic curves over the rationals. It incorporates code for computing the Mordell-Weil rank and group of an elliptic curve based on John Cremona's `mwrank` program. There are also facilities for computing heights of points, rational points on curves, and a database of curves with conductor less than 5077.

- Models: Weierstrass form, integral model, minimal model;
- Height, local height, naive height, canonical height, height pairing;
- Invariants for integral curves: conductor, regulator, Mordell-Weil rank, Tamagawa numbers;
- Mordell-Weil group, torsion subgroup;
- Kodaira symbols;
- Database of curves with conductor less than 5077, with tables of properties.

21.4.1 Example: Minimal Model and Torsion Points

We define an elliptic curve E over the rational field, then calculate its global minimal model M and the torsion points of E and M.

```
> E := EllipticCurve([-27, 55350]);
> E;
Elliptic Curve defined by y^2 = x^3 - 27*x + 55350 over
Rational Field
> M := MinimalModel(E);
> M;
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + 1
over Rational Field

> G, h := TorsionSubgroup(E);
> torsion_pts_E := [ h(g) : g in G ];
> torsion_pts_E;
[ (0, 1, 0), (-21, 216, 1), (51, -432, 1), (51, 432, 1),
(-21, -216, 1) ]

> G2, h2 := TorsionSubgroup(M);
> torsion_pts_M := [ h2(g) : g in G2 ];
> torsion_pts_M;
[ (0, 1, 0), (-1, 1, 1), (1, -3, 1), (1, 1, 1), (-1, -1, 1) ]
```

Since M is the minimal model of E, the two curves are isomorphic. We check this, assigning the isomorphism $M \to E$ to the identifier *iso*. We next show that *iso* gives the expected correspondence between the two sets of torsion points.

```
> is_iso, iso := IsIsomorphic(M, E);
> if not is_iso then
>     print "Something is badly wrong";
> else
>     print [iso(P) : P in torsion_pts_M] eq torsion_pts_E;
> end if;
true
```

21.4.2 Example: Integral points and Mordell-Weil group

We calculate the integral points on the elliptic curve F defined by $y^2 = x^3 + 17$ over the rational field. Firstly, we calculate a basis a, b for the integral points, by mapping the generators of the full Mordell-Weil group of F back to F. We then calculate $ia + jb$ for small values of i and j, in order to find other integral points.

```
> F := EllipticCurve([0, 17]);
> F;
Elliptic Curve defined by y^2 = x^3 + 17 over Rational Field
> MW, f := MordellWeilGroup(F);
> MW;
Abelian Group isomorphic to Z + Z
Defined on 2 generators (free)
> a := f(MW.1);
> b := f(MW.2);
> a, b;
(-2, 3, 1) (-1, 4, 1)
> intpts := [ pt : i, j in [-4..4] |
```

```
>     IsIntegral(pt) where pt is i*a + j*b] ;
> intpts;
[ (43, -282, 1), (5234, -378661, 1), (2, -5, 1), (8, 23, 1),
(4, 9, 1), (-2, -3, 1), (52, -375, 1), (-1, -4, 1), (-1, 4, 1),
(52, 375, 1), (-2, 3, 1), (4, -9, 1), (8, -23, 1), (2, 5, 1),
(5234, 378661, 1), (43, 282, 1) ]
> #intpts;
16
```

This agrees with Silverman, [J. H. Silverman, *The Arithmetic of Elliptic Curves* (New York: Springer-Verlag, 1986), 60.], who reports 16 integral points on F.

21.5 Elliptic Curves over a Finite Field

MAGMA contains several special functions for computing with elliptic curves defined over a finite field. The most elaborate of these is an implementation of the Schoof-Elkies-Atkin algorithm for calculating the order of a curve over a field of large characteristic. Lercier's extension of the algorithm is also implemented to enable calculation of the order of a curve over a field of characteristic two. In intermediate characteristic, the algorithm of Atkin carries out the calculation.

- Simplified models of elliptic curves
- Schoof-Elkies-Atkin algorithm for counting points in large characteristic and characteristic two
- Trace of Frobenius
- Supersingular testing
- Enumeration of points (for small fields)
- Random points
- Quadratic twist
- Discriminant of the endomorphism ring

21.5.1 Example: Endomorphism Ring

This example computes the endomorphism ring of an elliptic curve over a finite field.

```
> E := EllipticCurve([GF(239) | 2, 5]);
> IsSupersingular(E);
false
> Trace(E)^2 - 4*#BaseRing(E);
-947
> Factorization($1);
[ <947, 1> ]
```

There is no square factor, so the ring generated by the Frobenius map is maximal. Since the curve is ordinary, this equals the full endomorphism ring.

```
> E := EllipticCurve([GF(239) | 27, 7]);
> IsSupersingular(E);
false
> Trace(E)^2 - 4*#BaseRing(E);
-931
> Factorization($1);
[ <7, 2>, <19, 1> ]
```

The discriminant contains a square factor, so the endomorphism ring may or may not be that generated by the Frobenius. To find out we look at the 7-isogenies.

```
> f<x> := DivisionPolynomial(E,7);
> f;
7*x^24 + 128*x^22 + 217*x^21 + 44*x^20 + 136*x^19 + 216*x^18 + 68*x^17
    + 23*x^16 + 120*x^15 + 130*x^14 + 43*x^13 + 179*x^12 + 184*x^11 +
    97*x^10 + 190*x^9 + 14*x^8 + 119*x^7 + 112*x^6 + 69*x^5 + 2*x^4 +
    230*x^3 + 93*x^2 + 7*x + 163
```

How many rational 7-isogenies are there?

```
> Factorization(f);
[
    <x + 27, 1>,
    <x + 64, 1>,
    <x + 210, 1>,
    <x^7 + 93*x^6 + 76*x^5 + 113*x^4 + 93*x^3 + 217*x^2 + 175*x + 35, 1>,
    <x^7 + 134*x^6 + 141*x^5 + 82*x^4 + 93*x^3 + 121*x^2 + 202, 1>,
    <x^7 + 189*x^6 + 30*x^5 + 157*x^4 + 93*x^3 + 68*x^2 + 115*x + 222, 1>
]
```

Only one. A kernel polynomial for an isogeny of prime degree ℓ has degree $(\ell - 1)/2$, so in this case a kernel is defined by a degree three polynomial, which can only come from the product of the three linear factors. We can now conclude that the endomorphism ring is non-maximal at 7, so is generated by the Frobenius endomorphism. Let's find the isogeny:

```
> psi := &*[ $1[i][1] : i in [1..3] ];
> psi;
x^3 + 62*x^2 + 45*x + 78
> E1, I := IsogenyFromKernel(E, psi);
> E1;
Elliptic Curve defined by y^2 = x^3 + 58*x + 188 over GF(239)
```

Check they are isogenous and not isomorphic

```
> #E1;
```

```
235
> #E;
235
> IsIsomorphic(E1,E);
false
```

How many 7-isogenies are there from E1?

```
> Factorization(DivisionPolynomial(E1,7));
[
    <x + 10, 1>,
    <x + 12, 1>,
    <x + 13, 1>,
    <x + 14, 1>,
    <x + 19, 1>,
    <x + 32, 1>,
    <x + 35, 1>,
    <x + 43, 1>,
    <x + 46, 1>,
    <x + 65, 1>,
    <x + 66, 1>,
    <x + 84, 1>,
    <x + 89, 1>,
    <x + 102, 1>,
    <x + 120, 1>,
    <x + 138, 1>,
    <x + 151, 1>,
    <x + 156, 1>,
    <x + 162, 1>,
    <x + 192, 1>,
    <x + 198, 1>,
    <x + 208, 1>,
    <x + 211, 1>,
    <x + 224, 1>
]
```

All eight of them are rational! (Moreover each of the eight kernel polynomials of degree three splits into linear factors.) One isogeny is the dual of I, the other seven will lead to different curves. We now know that $E1$ has the largest possible endomorphism ring, of discriminant -19. Indeed, this discriminant has class number one, so its j-invariant is defined over the integers:

```
> jInvariant((1 + Sqrt(-19))/2);
-884735.99999999999999999999964265797 - 1.3213846776 E-22*i
> Round(Real($1));
-884736
> BaseRing(E1)!$1;
42
> jInvariant(E1);
42
```

The isogeny which generates the endomorphism ring will be the multiplication-by-5 map.

```
> Factorization(DivisionPolynomial(E1, 5));
[
    <x + 1, 1>,
    <x + 7, 1>,
    <x + 204, 1>,
    <x + 226, 1>,
    <x^2 + 121*x + 193, 1>,
    <x^2 + 176*x + 176, 1>,
    <x^2 + 225*x + 50, 1>,
    <x^2 + 235*x + 204, 1>
]
```

We have to decide which of the pairwise groupings of linear factors define kernel polynomials.

```
> #E1;
235
> P := (235 div 5)*Random(E1);
> { P, 2*P };
{ (13, 82, 1), (35, 219, 1) }
```

So one kernel polynomial disappears on 13 and 35; the third and fourth factors of the division polynomial. The first and second form the other pair.

```
> fact := Factorization(DivisionPolynomial(E1, 5));
> psi1 := fact[1][1] * fact[2][1];
> E2, J2 := IsogenyFromKernel(E1, psi1);
> _, h := IsIsomorphic(E2, E1);
> J := J2*h;
```

J is a generator of the endomophism ring of `E1`, which satifies either the minimal polynomial $J^2 - J + 5$, or $J^2 + J + 5$ of discriminant -19.

```
> F := FrobeniusMap(E1);
> P := Random(E1); P;  F(P) eq P;
(40, 117, 1)
true
> J(J(P)) - J(P) + 5*P;
(115, 146, 1)
> J(J(P)) + J(P) + 5*P;
(0, 1, 0)
```

So the endomorphism ring of `E1` is isomorphic to $\mathbf{Z}[J]/(J^2 + J + 5)$!

21.6 Databases for Elliptic Curves

MAGMA has several several databases specifically for elliptic curves.

21.6.1 John Cremona's database

For the convenience of users, MAGMA comes with a copy of John Cremona's elliptic curve database already installed. This database contains minimal models of all the elliptic curves over the rationals with conductor less than 5300, as well as a set of invariants for each. The database can be searched by the user, enabling quick selection of appropriate examples and counter-examples.

21.6.2 Modular Equations

MAGMA contains a database of modular equations calculated by Oliver Atkin. These equations play a central role in the point-counting algorithms for elliptic curves over finite fields, and are also directly available to the user. The database contains all modular equations for integers between 23 and 700, with the exception of those for 37 and 43. A supplementary database of canonical modular equations to replace the missing equations in the database is also available on request.

Chapter 22

Enumerative Combinatorics

22.1 The Enumeration Functions

In addition to algebraic structures such as power series rings, MAGMA has the following facilities for enumeration.

- Factorial
- Binomial, multinomial coefficients
- Stirling numbers of the first and second kind
- Fibonacci numbers, Bernoulli numbers, Harmonic numbers, and Eulerian numbers
- Number of partitions of n
- Enumeration of restricted and unrestricted partitions
- Sets of subsets, multisets, and subsequences of sets
- Permutations of sets (as sequences)

22.1.1 The Change Problem

How can we make change for sixty cents using five, ten, twenty and fifty cent coins? The **RestrictedPartitions** function will construct the ways for us.

```
> coins := {5, 10, 20, 50};
> RestrictedPartitions(60, coins);
[
  [ 50, 10 ],
  [ 50, 5, 5 ],
  [ 20, 20, 20 ],
  [ 20, 20, 10, 10 ],
  [ 20, 20, 10, 5, 5 ],
  [ 20, 20, 5, 5, 5, 5 ],
  [ 20, 10, 10, 10, 10 ],
  [ 20, 10, 10, 10, 5, 5 ],
  [ 20, 10, 10, 5, 5, 5, 5 ],
  [ 20, 10, 5, 5, 5, 5, 5, 5 ],
  [ 20, 5, 5, 5, 5, 5, 5, 5, 5 ],
  [ 10, 10, 10, 10, 10, 10 ],
```

```
  [ 10, 10, 10, 10, 10, 5, 5 ],
  [ 10, 10, 10, 10, 5, 5, 5, 5 ],
  [ 10, 10, 10, 5, 5, 5, 5, 5, 5 ],
  [ 10, 10, 5, 5, 5, 5, 5, 5, 5, 5 ],
  [ 10, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ],
  [ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
]
```

If we wish to know the number of different ways we can make change for varying amounts of money we can use the well-known generating function solution.

```
> P<x> := PowerSeriesRing(Rationals(), 101);
> gf := &*[1/(1-x^i) : i in coins];
> gf;
1 + x^5 + 2*x^10 + 2*x^15 + 4*x^20 + 4*x^25 + 6*x^30 + 6*x^35 + 9*x^40 + 9*x^45 +
  13*x^50 + 13*x^55 + 18*x^60 + 18*x^65 + 24*x^70 + 24*x^75 + 31*x^80 + 31*x^85 +
  39*x^90 + 39*x^95 + 49*x^100 + O(x^101)
```

We check that there are 18 ways to make change for sixty cents and find out that there are 49 ways to change a dollar with these coins.

22.1.2 Generation of Strings from a Grammar

Balanced strings of left and right parentheses may be generated using the grammar

$$S \to (S)S \mid \text{eps}.$$

For instance, there are 5 strings with 3 pairs of parentheses:

$$((())), (())(), ()(()), (()()), ()()().$$

The procedure below, by Don Taylor, generates balanced strings from the grammar. The probability of $(S)S$ being chosen rather than eps has been set to $\frac{3}{10}$.

```
 produce := procedure();
     seq := ["S"];
     rhs := ["(", "S", ")", "S"];
     i := 1;
     repeat
         if Random(1, 10) gt 7 then
             Insert(~seq, i, i, rhs);
         else
             Remove(~seq, i);
         end if;
         print #seq gt 0 select &*seq else "eps";
         i := Position(seq, "S");
     until i eq 0;
     print "Length:", #seq;
 end procedure;

> produce();
```

```
(S)S
()S
()(S)S
()((S)S)S
()(()S)S
()(())S
()(())(S)S
()(())()S
()(())()
Length: 8
> produce();
eps
Length: 0
> produce();
(S)S
((S)S)S
(()S)S
(())S
(())
Length: 4
```

To find out how many strings of length n this grammar produces we consider the ordinary generating function $f(x)$ of strings by length. Since the grammar is unambiguous we can read from the grammar that this power series satifies

$$f(x) = x^2 f(x)^2 + 1.$$

Solving the quadratic equation gives

$$f(x) = \frac{1 - \sqrt{1 - 4x^2}}{2x^2}.$$

To get a look at the first terms of this series we proceed as follows:

```
> P<x> := PowerSeriesRing(Rationals());
> f := (1 - Sqrt(1 - 4*x^2))/(2*x^2);
> f;
1 + x^2 + 2*x^4 + 5*x^6 + 14*x^8 + 42*x^10 + 132*x^12 + 429*x^14 + 1430*x^16 +
  O(x^18)
```

We find that the grammar produces 132 different strings of length 12. We easily identify the sequence of coefficients as the Catalan numbers.

22.2 Computing the Bernoulli Number B_{10000}

In this example we show how Magma can compute the Bernoulli number B_{10000}. The computation depends on a new asymptotically fast polynomial division algorithm of Daniel Ford, coupled with FFT methods for integer and polynomial multiplication. The whole computation takes about 14 hours on a 250MHz Ultrasparc.

The exponential generating function for the Bernoulli numbers is:

$$E(x) = \frac{x}{e^x - 1}.$$

This means that the i-th Bernoulli number B_i is $i!$ times the coefficient of x^i in $E(x)$. The Bernoulli numbers $B_0, \ldots, B_n$ for any n can thus be calculated by computing the above power series and scaling the coefficients.

First we illustrate the method for a small number of coefficients (printing out the computed power series at each stage).

For the small example, we first set the final precision p we desire to be 14. We then create the denominator $D = e^x - 1$ of the exponential generating function to precision $p + 1$ (we need $p + 1$ since we lose precision when we divide by the denominator and the valuation changes).

```
> p := 14;
> S<x> := PowerSeriesRing(RationalField());
> D := Exp(x + O(x^(p + 2))) - 1;
> D;
x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + 1/720*x^6 + 1/5040*x^7 +
    1/40320*x^8 + 1/362880*x^9 + 1/3628800*x^10 + 1/39916800*x^11 +
    1/479001600*x^12 + 1/6227020800*x^13 + 1/87178291200*x^14 +
    1/1307674368000*x^15 + O(x^16)
```

We then form the quotient $E = x/D$ which gives the exponential generating function.

```
> E := x / D;
> E;
1 - 1/2*x + 1/12*x^2 - 1/720*x^4 + 1/30240*x^6 - 1/1209600*x^8 + 1/47900160*x^10
    - 691/1307674368000*x^12 + 1/74724249600*x^14 + O(x^15)
```

We finally compute the Laplace transform of E (which multiplies the coefficient of x^n by $n!$) to yield the generating function of the Bernoulli numbers to precision p. Thus the coefficient of x^{14} here is B_{14}.

```
> B := Laplace(E);
> B;
1 - 1/2*x + 1/6*x^2 - 1/30*x^4 + 1/42*x^6 - 1/30*x^8 + 5/66*x^10 - 691/2730*x^12
    + 7/6*x^14 + O(x^15)
```

Now for the giant computation! We do the exactly the same but with $p = 10000$. First, computing $e^x - 1$ to precision $p + 1$ involves computing $n!$ for $n = 0, \ldots 10001$ which is a fairly small part of the whole computation!

```
> p := 10000;
> S<x> := PowerSeriesRing(RationalField());
> time D := Exp(x + O(x^(p + 2))) - 1;
Time: 10.129
```

Forming the quotient $E = x/D$ is where the bulk of the effort for the computation is spent. This uses the fast division algorithm, coupled with the FFT multiplication methods. Without these methods (i.e., using classical quadratic-complexity methods), the computation would probably take weeks.

```
> time E := x / D;
Time: 45209.089
```

As before, we compute the Laplace transform of E to yield the generating function B. This takes a while because the denominators of the coefficients of E are large and GCDs must be taken of these with the appropriate factorials.

```
> time B := Laplace(E);
Time: 4297.339
```

Printing the resulting generating function B takes about 70MB (900000 lines) of output, so we will not include the output here! The printing actually takes about 5000 seconds because of all the necessary base conversion from binary to decimal when printing the coefficients.

We note that the numerator of B_{10000} has 27706 decimal digits.

```
> // Set x to B_{10000}
> C := Coefficients(B);
> x := C[10001]; // sequence root at index 1
> Denominator(x);
2338224387510
> time s := Sprint(x);
Time: 2.829
> #s;
27706
```

The Bernoulli numbers B_j satisfy the following recurrence relation:

$$\sum_{j=0}^{m} \binom{m+1}{j} B_j = 0.$$

This recurrence relation is commonly used to define the Bernoulli numbers; one could use it to compute B_{10000} (with rational arithmetic) but this would be incredibly slow and the point of the example is to use a better method!

We check our result by evaluating the left-hand side of the recurrence relation, with all computations done modulo a large prime p (which is the smallest prime greater than 10^{30})—the result should be zero modulo p.

```
> // Compute Binomial(n, k) in ring R:
> bin := func<n, k, R | &*[R|x: x in [n..n-k+1 by -1]]/&*[R|x: x in [2..k]]>;

> // Return B_i using coefficient sequence C:
> bern := func<i| C[i + 1]>;

> // Create ring R of integers modulo p
> p := NextPrime(10^30);
> p;
1000000000000000000000000000057
> R := IntegerRing(p);

> // Evaluate LHS of recurrence relation:
> m:=10000;
> time &+[bin(m + 1, j, R) * R!bern(j): j in [0 .. m]];
0
Time: 830.579
```

Finally, we make some comments on the actual run of Magma which was used to generate this example.

The largest amount of time taken for the computation was the forming of the product of two polynomials with integer coefficients, each of degree 5000. The coefficients of both polynomials were each about 15000–20000 decimal digits on average! (This product arose within the recursive division algorithm.)

To form the product, each polynomial was transformed using the FFT algorithm into a vector of length 16384 whose points were each integer blocks of 131072 bits each. Each FFT transform took about 96 seconds and the space taken by each vector was about 270MB of memory.

The vectors were then multiplied pointwise (placing the result back into the first vector to save memory). This involved 16384 products of pairs of integers, each of which was 131072 bits (about 40000 decimal digits) long. These products were in turn done by the Schönhage-Strassen FFT integer multiplication algorithm (so the FFT method was genuinely called recursively)! Each integer product took 0.085 seconds, giving a total time of 1390 seconds for the 16384 products.

Finally, the inverse FFT was applied to the resulting product vector to obtain the product of the original polynomials. This FFT transform took 94 seconds, and, adding in the time for a few trivial house-keeping operations, the total time for the polynomial product was 1700 seconds.

Chapter 23

Graphs

23.1 Construction and Properties of Graphs

Graphs may be directed or undirected. In addition, their vertices and/or their edges may be labelled.

- Directed and undirected graphs
- Optional vertex and edge labels
- Operations: union, join, product, contraction, switching etc
- Standard graphs: complete, complete bipartite, k-cube
- Properties: connected, regular, eulerian
- Algebraic invariants: Characteristic polynomial, spectrum
- Diameter, girth, circumference
- Cut-vertices, maximal independent sets, cliques
- Chromatic number, chromatic index, chromatic polynomial

23.1.1 Construction of Tutte's 8-cage

The following construction of Tutte's 8-cage uses the technique described in P. Lorimer, *J. of Graph Theory*, **13**, 5 (1989), 553–557. The graph is constructed so that it has $G = P\Gamma L(2, 9)$ in its representation of degree 30 as its automorphism group. The vertices of the graph correspond to the points on which G acts. The neighbours of vertex 1 are the points lying in the unique orbit N_1 of length 3 of the stabilizer of 1. The edges for vertex i are precisely the points N_1^g where g is an element of G such that $1^g = i$.

```
> G := PermutationGroup< 30 |
>     (1, 2)(3, 4)(5, 7)(6, 8)(9, 13)(10, 12)(11, 15)(14, 19)(16, 23)
>         (17, 22)(18, 21)(20, 27)(24, 29)(25, 28)(26, 30),
>     (1, 24, 28, 8)(2, 9, 17, 22)(3, 29, 19, 15)(4, 5, 21, 25)
>         (6, 18, 7, 16)(10, 13, 30, 11)(12, 14)(20, 23)(26, 27) >;
> N1 := rep{ o : o in Orbits(Stabilizer(G, 1)) | #o eq 3 };
> tutte := Graph< 30 | <1, N1>^G >;
```

23.1.2 Construction of the Grötzch Graph

The Grötzch graph may be built by taking the complete graph $K5$, choosing a cycle of length 5 (say, 1-3-5-2-4), inserting a vertex of degree two on each chord of this cycle, and finally connecting each of these vertices to a new vertex.

```
> G := CompleteGraph(5);
> E := EdgeSet(G);
> H := InsertVertex({ E | { 1, 3 }, { 1, 4 }, { 2, 4 }, { 2, 5 }, { 3, 5 } });
> L,V := Union(H, CompleteGraph(1));
> L := L + { { V.11, V.6 }, { V.11, V.7 }, { V.11, V.8 }, { V.11, V.9 },
>             { V.11, V.10 } };
> L;

Graph
Vertex  Neighbours

1       2 5 6 7 ;
2       1 3 8 10 ;
3       2 4 6 9 ;
4       3 5 7 8 ;
5       1 4 9 10 ;
6       1 3 11 ;
7       1 4 11 ;
8       2 4 11 ;
9       3 5 11 ;
10      2 5 11 ;
11      6 7 8 9 10 ;
```

23.2 Automorphism Groups

Brendan McKay's automorphism program (`nauty`) is used for computing automorphism groups.In accordance with the Magma philosophy, a graph may be studied under the action of an automorphism group. Using the G-set mechanism an automorphism group can be made to act on any desired set of objects derived from the graph.

- Graphs from groups: Cayley graph, orbital graph
- Automorphism group (B. McKay's algorithm), canonical labelling
- Testing of pairs of graphs for isomorphism
- Group actions on a graph: orbits and stabilizers of vertex and edge sequences
- Symmetry properties: vertex transitive, edge transitive, k-arc transitive, distance transitive, distance regular
- Intersection numbers of a distance regular graph

23.2.1 Automorphism Group of the 8-dimensional Cube Graph

We illustrate the use of some of the automorphism group functions on the graph of the 8-dimensional cube.

```
> g := KCubeGraph(8);
> aut := AutomorphismGroup(g);
> Order(aut), FactoredOrder(aut);
10321920 [ <2, 15>, <3, 2>, <5, 1>, <7, 1> ]
> CompositionFactors(aut);
    G
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Alternating(8)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
> IsVertexTransitive(g);
true
> IsEdgeTransitive(g);
true
> IsSymmetric(g);
true
> IsDistanceTransitive(g);
true
> IntersectionArray(g);
[ 8, 7, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 7, 8 ]
```

Chapter 24

Incidence Structures and Designs

24.1 Construction nand Properties

General incidence structures provide a universe in which families of incidence structures satisfying stronger conditions (linear spaces, t-designs, etc) reside.

- Creation of a general incidence structure, near-linear space, linear space, design
- Difference sets: standard difference sets, development
- Hadamard designs, Witt designs
- Unary operations: complement, contraction, dual, residual
- Binary operations: sum, union
- Invariants for an incidence structure: point degrees, block degrees, covalence
- Invariants for a design: replication number, order, covalence, intersection numbers, Pascal triangle
- Properties: balanced, complete, uniform, self-dual, simple, Steiner
- Near-linear space operations: connection number, point and line regularity, restriction
- Resolution of a design
- Graphs and codes from designs: block graph, incidence graph, point graph, linear code

Tools are provided for constructing designs from difference sets, Hadamard matrices, codes and other designs. The standard families of difference sets are incorporated. A major feature is the ability to compute automorphism groups and to test pairs of incidence structures for isomorphism.

24.1.1 The Witt Design and its Resolution

We construct the Witt design D on 12 points, together with its point-set P and block-set B.

```
> D, P, B := WittDesign(12);
> D;
5-(12, 6, 1) Design with 132 blocks
```

This design has a resolution; i.e. the block-set of D can be partitioned into parallel classes. In this example, the automorphism group A acts transitively on the classes of the resolution. Hence we can get the resolution of D by taking the orbit of a single parallel class under A. To construct the initial class, we select one of the blocks of points and then find the block consisting of the remaining points.

```
> b1 := B.1; // first block (arbitrary choice)
> b2 := B ! (Set(P) diff b1);
> class := {b1, b2};
> class;
{ {1, 6, 7, 8, 11, 12}, {2, 3, 4, 5, 9, 10} }
> A, PP, BB := AutomorphismGroup(D);
> resolution := Orbit(A, PP, class);
> #resolution;
66
```

24.2 Automorphism Groups

- Automorphism group (J. Leon's algorithm), isomorphism testing
- Group actions on a design: orbits and stabilizers of points and blocks
- Symmetry properties: point transitive, block transitive

24.2.1 Automorphism Group of $\mathrm{PG}(2,2)$

We construct the automorphism group of the block design corresponding to the projective plane $\mathrm{PG}(2,2)$.

```
> B1 := { 1, 2, 4 };
> B2 := { 2, 3, 5 };
> B3 := { 3, 4, 6 };
> B4 := { 4, 5, 7 };
> B5 := { 5, 6, 1 };
> B6 := { 6, 7, 2 };
> B7 := { 7, 1, 3 };
> D := Design< 2, 7 | B1, B2, B3, B4, B5, B6, B7 >;
> D;
2-(7, 3, 1) Design with 7 blocks
> A, PP, BB := AutomorphismGroup(D);
> A;
Permutation group A acting on a set of cardinality 7
Order = 168 = 2^3 * 3 * 7
    (1, 2)(3, 5, 6, 7)
    (2, 5, 3)(4, 6, 7)
    (2, 4)(3, 5, 7, 6)
    (3, 6)(5, 7)
    (3, 5)(6, 7)
> PP;
GSet{ 1, 2, 3, 4, 5, 6, 7 }
> BB;
GSet{ {1, 2, 4}, {2, 3, 5}, {3, 4, 6}, {4, 5, 7}, {1, 5, 6},
{2, 6, 7}, {1, 3, 7} }
```

The returned values *PP* and *BB* are A-sets corresponding to the action of the automophism group on the points and blocks of D respectively. Using them, we can determine the stabilizer of the

block *B1* in two ways—first, by regarding *B1* as a set of points to be stabilized under the action of A on the points; and second, by regarding *B1* as a block to be fixed under the action of A on the blocks.

```
> StabB1_1 := Stabilizer(A, PP, B1);
> StabB1_2 := Stabilizer(A, BB, B1);
> StabB1_1;
Permutation group StabB1_1 acting on a set of cardinality 7
Order = 24 = 2^3 * 3
    (3, 7)(5, 6)
    (2, 4)(5, 6)
    (1, 2)(5, 7)
> StabB1_1 eq StabB1_2;
true
```

24.2.2 Constructing a Design from a Difference Set

The function `DifferenceSet` given below may be used to construct the design corresponding to a given set S of integers forming a perfect difference set modulo n.

```
> DifferenceSet := func< S, n |
>     [ { (x + i) mod n + 1 : x in S } : i in [0..n-1] ] >;
```

We use this function to construct a $(19, 9, 4)$-design and then proceed to compute its automorphism group.

```
> DSet := {1, 4, 5, 6, 7, 9, 11, 16, 17};
> D := Design< 2, 19 | DifferenceSet(DSet, 19) >;
> D;
2-(19, 9, 4) Design with 19 blocks
> Blocks(D);
{@
{2, 5, 6, 7, 8, 10, 12, 17, 18}, {3, 6, 7, 8, 9, 11, 13, 18, 19},
{1, 4, 7, 8, 9, 10, 12, 14, 19}, {1, 2, 5, 8, 9, 10, 11, 13, 15},
{2, 3, 6, 9, 10, 11, 12, 14, 16}, {3, 4, 7, 10, 11, 12, 13, 15, 17},
{4, 5, 8, 11, 12, 13, 14, 16, 18}, {5, 6, 9, 12, 13, 14, 15, 17, 19},
{1, 6, 7, 10, 13, 14, 15, 16, 18}, {2, 7, 8, 11, 14, 15, 16, 17, 19},
{1, 3, 8, 9, 12, 15, 16, 17, 18}, {2, 4, 9, 10, 13, 16, 17, 18, 19},
{1, 3, 5, 10, 11, 14, 17, 18, 19}, {1, 2, 4, 6, 11, 12, 15, 18, 19},
{1, 2, 3, 5, 7, 12, 13, 16, 19}, {1, 2, 3, 4, 6, 8, 13, 14, 17},
{2, 3, 4, 5, 7, 9, 14, 15, 18}, {3, 4, 5, 6, 8, 10, 15, 16, 19},
{1, 4, 5, 6, 7, 9, 11, 16, 17}
@}
> Aut := AutomorphismGroup(D);
> A := OrbitImage(Aut, 1);
> A;
Permutation group A acting on a set of cardinality 19
Order = 171 = 3^2 * 19
    (1, 4, 12, 8, 10, 9, 19, 14, 7)(2, 13, 17, 15, 16, 6, 11, 18, 5)
    (2, 5, 17, 8, 10, 18, 12, 7, 6)(3, 9, 14, 15, 19, 16, 4, 13, 11)
```

```
> IsPrimitive(A);
true
> IsFrobenius(A);
true
```

Thus the group is a Frobenius group and the normal subgroup of order 19 must be the Frobenius kernel.

Chapter 25

Finite Planes

25.1 Construction and Basic Properties

Although finite planes correspond to particular families of designs, separate categories are provided for both projective and affine planes in order to exploit the rich structure possessed by these objects.

- Creation of classical and non-classical finite projective and affine planes
- Subplanes, dual of a projective plane
- Numerical invariants: order, p-rank
- Properties: Desarguesian, self-dual
- Parallel classes of an affine plane
- k-arcs: testing, complete, tangents, secants, passants
- Conics: through given points, knot, exterior, interior
- Unitals: testing, tangents, feet
- Affine to projective planes and vice versa
- Related structures: design, incidence matrix, incidence graph, linear code
- Collineation group, isomorphism testing (optimized algorithm for projective planes)
- Central collineations: testing, groups
- Group actions on a plane: orbits and stabilizers of points and lines
- Symmetry properties: point transitive, line transitive

Apart from elementary invariants, a reasonably fast method is available for testing whether a plane is desarguesian. Among special configurations of interest, a search procedure for k-arcs is provided. A specialized algorithm developed by Jeff Leon is used to compute the collineation group of a projective plane while the affine case is handled by the incidence structure method. The collineation group (order $2^3 3^8$) of a "random" projective plane of order 81 supplied by Gordon Royle was found in 1 202 seconds. As with graphs and designs the G-set mechanism gives the action of the collineation group on any appropriate set.

25.1.1 Hermitian Unital

This example is partly derived from J. D. Key's paper "Some applications of Magma in designs and codes: oval designs, hermitian unitals and generalized Reed-Muller codes" (1995, unpublished

manuscr ipt). A unital (or unitary) design is a Steiner 2-design with parameters 2-$(m^3+1, m+1, 1)$. In the context of the desarguesian plane of square order q^2, the set of absolute points and non-absolute lines of a unitary polarity forms a unital known as the *hermitian unital*.

Given a prime p and an integer m, the function below returns two values: the hermitian unital $x^{q+1} + y^{q+1} + z^{q+1} = 0$ in $\mathrm{PG}(2, q^2)$, where $q = p^m$; and the design whose blocks are those intersections of lines of the plane with the unital which have cardinality $q + 1$.

```
HUnital := function(p, m)
    q := p^m;
    P, V, L := ProjectivePlane(q ^ 2);
    hu := { pt : pt in V |
        IsZero(pt[1]^(q+1) + pt[2]^(q+1) + pt[3]^(q+1)) };
    blks := [blk : lin in L | #blk eq (q+1) where blk is lin meet hu];
    return hu, Design< 2, SetToIndexedSet(hu) | blks >;
end function;
```

We now evaluate the function for the case $p = 2, m = 2$, and compute the automorphism group for the resulting design:

```
> herm, D := HUnital(2, 2);
> IsUnital(herm);
true
> D;
2-(65, 5, 1) Design with 208 blocks
> IsSteiner(D, 2);
true
> A := AutomorphismGroup(D);
> Order(A);
249600
> CompositionFactors(A);
    G
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  2A(2, 4)               = U(3, 4)
    1
```

25.2 Construction of an Affine Plane by Derivation

We construct the collineation group $\mathrm{PGL}(3, q)$ in its action on the points of $\mathrm{PG}(2, q)$, for the case where $q = 16$.

We begin by creating the projective plane *PP* from the finite field with 16 elements, along with the point-set and line-set of the plane. Then the collineation group G is formed.

```
> q := 16;
> F<w> := FiniteField(q);
> PP, Pts, Lns := ProjectivePlane(F);
```

```
> PP;
The projective plane PG(2, 16)
> #Pts, #Lns;
273 273

> G, gspt, gsln := CollineationGroup(PP);
> G;
Permutation group G acting on a set of cardinality 273
Order = 2^14 * 3^3 * 5^2 * 7 * 13 * 17
```

We next define a certain Hall oval, and print its points. We choose points P and Q on the oval and find the line PQ that contains P and Q. Then we choose a point X not on the line PQ: this is done by choosing another point on the oval.

```
> oval := { Pts | [1, x, w^4*x^14 + w^24*x^12 + w^12*x^10 + w^18*x^8
>   + w^10*x^6 + w^10*x^4 + w^12*x^2] : x in F }
>   join { Pts | [0,1,0], [0,0,1] };
> oval;
{ ( 1 : 0 : 0 ), ( 1 : w^4 : w^3 ), ( 1 : w : w^4 ), ( 0 : 1 : 0 ),
( 1 : w^5 : w^10 ), ( 1 : w^14 : w ), ( 0 : 0 : 1 ), ( 1 : w^10 : w^7 ),
( 1 : w^13 : w^11 ), ( 1 : w^7 : w^8 ), ( 1 : w^11 : w^2 ),
( 1 : w^12 : w^6 ), ( 1 : w^9 : w^5 ), ( 1 : w^6 : w^14 ), ( 1 : w^8 : w^12 ),
( 1 : w^2 : w^9 ), ( 1 : 1 : 1 ), ( 1 : w^3 : w^13 ) }

> P := Rep(oval);
> Q := Rep(Exclude(oval, P));
> P, Q;
( 1 : 0 : 0 ) ( 0 : 1 : 0 )
> PQ := Lns![P, Q];
> PQ;
< 0 : 0 : 1 >

> X := Rep(oval diff {P, Q});
> X;
( 1 : w^4 : w^3 )
> XP := Lns![X, P];
> XQ := Lns![X, Q];
> XP, XQ;
< 0 : 1 : w > < 1 : 0 : w^12 >
```

Now we construct: the group $H1$ of central collineations with axis PQ; the group $H2$ of elations with centre P and axis PQ; the group $H3$ of homologies with centre P and axis XQ; and the group $H4$ of central collineations with centre P and axis through Q.

```
> H1 := Stabilizer(G, gspt, Setseq(Set(PQ)));
> H2 := SylowSubgroup(Stabilizer(H1, gsln, XP), 2);
> H3 := Stabilizer(Stabilizer(G, gspt, P), gspt, Setseq(Set(XQ)));
> H4 := sub< G | H2, H3 >;
```

We construct the set *afflines* containing the lines of the new plane. They are: the translates of *oval* (excluding P and Q) under the central collineations: the lines of $\mathrm{PG}(2, F)$ (excluding PQ)

incident with P; and the lines of $\mathrm{PG}(2, F)$ (excluding PQ) incident with Q. Then we can construct the affine plane itself.

```
> afflines := Orbit(H4, gspt, oval diff {P, Q}) join
>   { Exclude(Set(l), Y) : l in Lns, Y in {P,Q} | l ne PQ and Y in l };
> #afflines;
272
> affpts := &join afflines;
> #affpts;
256
> affpl := AffinePlane< SetToIndexedSet(affpts) | Setseq(afflines) >;
> affpl;
Affine plane of order 16
```

Finally, we check that the plane is desarguesian by calculating the p-rank, which equals the dimension of the corresponding linear code:

```
> C := LinearCode(affpl, PrimeField(F));
> Dimension(C);
81
```

Chapter 26

Error-correcting Codes

26.1 Construction and Basic Properties

Currently, the coding theory module is designed for linear codes over finite fields though this will be extended to linear codes over finite rings in the near future.

- Construction of linear codes as subspaces of vector spaces and modules
- Construction of general cyclic codes
- Construction of standard linear, cyclic, and alternant codes
- Combining codes: sum, intersection, (external) direct sum, Plotkin sum
- Combining codes: concatenation
- Modifying a code: augment, extend, expurgate, lengthen, puncture, shorten, etc
- Changing the alphabet: extend field, restrict field, subfield subcode, trace code
- Construction of basic codes: Hamming, Reed-Muller, Golay, QR
- Construction of BCH codes and their generalizations: Alternant, BCH, Goppa, Reed-Solomon, generalized Reed-Solomon, Srivastava, generalized Srivastava
- Construction of Justensen codes
- Standard form
- Construction of all information sets of a code
- Properties: cyclic, self-orthogonal, weakly self-orthogonal, perfect, etc.
- Syndrome decoding, Alternant decoding
- Bounds: upper and lower bounds on the cardinality of a largest code having given length and minimum distance
- Bounds: upper asymptotic bounds on the information rate

A large number of standard constructions are supported together with the determination of all the standard invariants.

26.1.1 Construction of a Goppa Code

We construct a Goppa code of length 31 over GF(2) with generator polynomial $G(z) = z^3 + z + 1$.

```
> q := 2^5;
> K<w> := FiniteField(q);
> P<z> := PolynomialRing(K);
> G := z^3 + z + 1;
> L := [w^i : i in [0 .. q - 2]];
> C := GoppaCode(L, G);
> C;
[31, 16, 7] Goppa code (r = 3) over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 1 1 0 0 0 0 1]
[0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 1 1 1 0 0 1 0 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 1]
[0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 1 0 1 0 1 0 1 0 1 1]
[0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 1 1 1 1 1]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 1 0 0 1 0 1 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0 0 1 1 1 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 1 0 1 0 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 1 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 1 0 1 1 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 1 0 1 0 0 1 1 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 0 1 1 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 1 1 1 1]
```

26.2 Minimum Distance and Weight Distribution

Algorithms are provided for computing the minimum weight and weight distribution. For example, computation of the minimum weight (15) of a $[79, 40, 15]$ quadratic-residue code takes 23 seconds; computation of the weight distribution of the $[64, 22, 16]$ Reed-Muller code ($r = 2, m = 6$) takes 2.9 seconds.

- Minimum weight, words of minimum weight
- Code words of designated weight
- Weight distribution, weight enumerator, MacWilliams transform
- Complete weight enumerator, MacWilliams transform
- Coset weight distribution, covering radius, diameter

26.2.1 Weight Distribution of a Reed-Muller Code

We determine the weight distribution and weight enumerator for the the second-order binary Reed-Muller code of length 128 and its dual.

```
> R := ReedMullerCode(2, 7);
> #R;
536870912
> MinimumWeight(R);
32
> time WeightDistribution(R);
[ <0, 1>, <32, 10668>, <48, 5291328>, <56, 112881664>, <64, 300503590>,
<72, 112881664>, <80, 5291328>, <96, 10668>, <128, 1> ]
Time: 426.939
>
> f<x, y> := WeightEnumerator(R);
> f;
x^128 + 10668*x^96*y^32 + 5291328*x^80*y^48 + 112881664*x^72*y^56 +
    300503590*x^64*y^64 + 112881664*x^56*y^72 + 5291328*x^48*y^80 +
    10668*x^32*y^96 + y^128
> D := Dual(R);
> #D;
6338253001141147007483516026 88
> WeightDistribution(D);
[ <0, 1>, <8, 188976>,
<12, 148157184>, <14, 5805342720>,
<16, 352501184760>, <18, 14090340827136>,
<20, 445990551166720>, <22, 11148730324353024>,
<24, 224814298345622160>, <26, 3704888469231108096>,
<28, 50486579825291883008>, <30, 574502111223143792640>,
<32, 5505259862572668584988>, <34, 4474863584391360577536 0>,
<36, 3104702958704068703851 52>, <38, 1848689416882328323358720>,
<40, 9492309127074743252712240>, <42, 4220274020877898748775 6288>,
<44, 1630560417353548433829648640>, <46, 5491916536309038087424 90112>,
<48, 1616902022777436781296463560>, <50, 4170947258549850556429074432>,
<52, 9445968792148616532912076032>, <54, 1881272610457063492103307 2640>,
<56, 3299556702044875730081668 0976>, <58, 5102036860250738031368365 6704>,
<60, 6961253682567332839539246 1824>, <62, 8385899464817855182050990 4896>,
<64, 8922497198992443834327614 4710>, <66, 8385899464817855182050990 4896>,
<68, 6961253682567332839539246 1824>, <70, 5102036860250738031368365 6704>,
<72, 3299556702044875730081668 0976>, <74, 1881272610457063492103307 2640>,
<76, 9445968792148616532912076032>, <78, 4170947258549850556429074432>,
<80, 1616902022777436781296463560>, <82, 5491916536309038087424 90112>,
<84, 1630560417353548433829648640>, <86, 4220274020877898748775 6288>,
<88, 9492309127074743252712240>, <90, 1848689416882328323358720>,
<92, 3104702958704068703851 52>, <94, 4474863584391360577536 0>,
<96, 5505259862572668584988>, <98, 574502111223143792640>,
<100, 50486579825291883008>, <102, 3704888469231108096>,
<104, 224814298345622160>, <106, 11148730324353024>,
<108, 445990551166720>, <110, 14090340827136>,
<112, 352501184760>, <114, 5805342720>,
<116, 148157184>, <120, 188976>,
<128, 1> ]
```

26.3 Syndrome Decoding

We construct a Hamming code C, encode an information word using C, introduce one error, and then decode by calculating the syndrome of the "received" vector and applying the `CosetLeaders` map to the syndrome to recover the original vector.

First we set C to be the third-order Hamming code over the finite field with two elements.

```
> C := HammingCode(GF(2), 3);
> C;
[7, 4, 3] Hamming code (r = 3) over GF(2)
Generator matrix:
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
```

Then we set L to be the set of coset leaders of C in its ambient space V and f to be the map which maps the syndrome of a vector in V to its coset leader in L.

```
> L, f := CosetLeaders(C);
> L;
{@
    (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 1 0 0),
    (0 0 0 0 0 1 0),
    (0 0 0 0 0 0 1)
@}
```

Since C has dimension 4, the degree of the information space I of C is 4. We set i to be an "information vector" of length 4 in I, and then encode i by C by setting w to be the product of i by the generator matrix of C.

```
> I := InformationSpace(C);
> I;
Full Vector space of degree 4 over GF(2)
> i := I ! [1, 0, 1, 1];
> w := i * GeneratorMatrix(C);
> w;
(1 0 1 1 0 1 0)
```

Now we set r to be the same as w but with an error in the 7th coordinate (so r is the "received vector").

```
> r := w;
> r[7] := 1;
```

```
> r;
(1 0 1 1 0 1 1)
```

Finally we let s be the syndrome of r with respect to C, apply f to s to get the coset leader l, and subtract l from r to get the corrected vector v. Finding the coordinates of v with respect to the basis of C (the rows of the generator matrix of C) gives the original information vector.

```
> s := Syndrome(r, C); s;
(1 1 1)
> l := f(s); l;
(0 0 0 0 0 0 1)
> v := r - l; v;
(1 0 1 1 0 1 0)
> res := I ! Coordinates(C, v); res;
(1 0 1 1)
```

26.4 Automorphism Group

Automorphism groups are be computed using Jeff Leon's PERM package.

- Automorphism groups of linear codes over GF(p) (prime p), GF(4)
- Testing pairs of codes for isomorphism over GF(p) (prime p), GF(4)
- Group actions on a code

26.4.1 Automorphism group of a Reed-Muller code

We compute the automorphism group of the second-order Reed-Muller code of length 64.

```
> C := ReedMullerCode(2, 6);
> aut := AutomorphismGroup(C);
> FactoredOrder(aut);
[ <2, 21>, <3, 4>, <5, 1>, <7, 2>, <31, 1> ]
> CompositionFactors(aut);
    G
    |  A(5, 2)                = L(6, 2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
```

26.4.2 The Cheng-Sloane Binary $[32, 17, 8]$ Code

In 1988, Cheng and Sloane discovered a binary $[32, 17, 8]$ code having twice as many codewords as the $[32, 16, 8]$ extended quadratic residue code. The code can be constructed as a submodule of the permutation module over GF(2) of the automorphism group of the 4-dimensional cube acting on edges. We construct the code and compute its weight enumerator and automorphism group.

```
> q4 := KCubeGraph(4); // as above
> G := EdgeGroup(q4); print Order(G);
384
> M := GModule(G, GaloisField(2));
> L := SubmoduleLattice(M); print #L;
373
```

Thus, the module M has 373 submodules. We now proceed to find all codes of minimum weight 8 that arise as 17-dimensional submodules of L.

```
> codes := { C : N in L |
>     Dimension(N) eq 17 and MinimumWeight(C) eq 8
>     where C is LinearCode(Morphism(N)) };
> print #codes;
1
```

Therefore, there is only one such code, the one discovered by Cheng and Sloane. We compute its weight enumerator and automorphism group.

```
> C := Rep(codes);
> Wt<x> := WeightEnumerator(C); print Wt;
x^32 + 908*x^24 + 3328*x^22 + 14784*x^20 + 27392*x^18 + 38246*x^16
+ 27392*x^14 + 14784*x^12 + 3328*x^10 + 908*x^8 + 1
> A := AutomorphismGroup(C); print A;
Permutation group A acting on a set of cardinality 32
Order = 384 = 2^7 * 3
    (1, 29, 2, 22)(3, 25, 4, 23)(5, 28, 18, 8)(6, 7, 17, 32)
        (9, 26, 14, 13)(10, 11, 12, 30)(15, 21, 16, 20)(19, 27, 24, 31)
    (1, 2)(3, 16)(4, 15)(5, 14)(6, 12)(7, 30)(8, 26)(9, 18)(10, 17)
        (11, 32)(13, 28)(19, 24)(20, 25)(21, 23)(22, 29)(27, 31)
    (1, 31)(2, 27)(3, 16)(4, 15)(5, 8)(6, 7)(9, 13)(10, 11)(12, 30)
        (14, 26)(17, 32)(18, 28)(19, 22)(20, 23)(21, 25)(24, 29)
    (5, 17)(6, 18)(7, 28)(8, 32)(9, 12)(10, 14)(11, 26)(13, 30)
        (19, 24)(20, 21)(22, 29)(23, 25)
    (2, 3)(6, 7)(9, 19)(10, 20)(11, 23)(12, 24)(13, 22)(14, 21)
        (16, 27)(18, 28)(25, 26)(29, 30)
    (3, 15)(4, 16)(5, 9)(6, 10)(7, 11)(8, 13)(12, 17)(14, 18)(20, 23)
        (21, 25)(26, 28)(30, 32)
```

Not unexpectedly, the automorphism group A of our code is isomorphic to our initial group, G!

26.4.3 Construction of 5-designs from Symmetry Codes

V. Pless (1972) constructed 5-designs from the codewords of minimum weight in a symmetry code.

The function `Paley(`q`)` below returns a $(q+1)$ by $(q+1)$ Paley matrix, i.e., a conference matrix of the form

$$\begin{pmatrix} 0 & 1 & 1 & \cdots & 1 \\ \pm 1 & & & & \\ \vdots & & & Q & \\ \pm 1 & & & & \end{pmatrix}$$

where Q is a q by q matrix of the form returned by `Qmat(`q`)`:

```
Z := IntegerRing();

Chi := func< x | x eq 0 select 0
                    else IsSquare(x) select 1
                    else -1 >;

Qmat := func< q | MatrixRing(Z, q) !
    [ Chi(Fseq[i] - Fseq[j]) : i, j in [1..q] ]
    where Fseq is
      ([0] cat Setseq(Exclude(Set(GF(q)), 0))) >;


ConstZMat := func< n, r, c |
    RMatrixSpace(Z, r, c) ! [ n : i in [1..r*c]] >;

Paley := func< q | VerticalJoin(
    HorizontalJoin(ConstZMat(0, 1, 1),
                   ConstZMat(1, 1, q)),
    HorizontalJoin(ConstZMat(pm1, q, 1)
      where pm1 is (q mod 4 eq 1 select 1 else -1),
                   Qmat(q))) >;
```

We create a ternary $[36, 18]$ symmetry code from the generator matrix $(I \mid C)$, where C is a Paley matrix with $q = 17$.

```
> q := 17;
> C := Paley(q);
> M := MatrixRing(GF(3), q+1);
> gen := HorizontalJoin(Identity(M), M ! C);

> LC := LinearCode(gen);
> LC : Minimal;
[36, 18] Linear Code over GF(3)
> #LC;
387420489
> LC eq Dual(LC);
true
```

```
> time minwt := MinimumWeight(LC);
Time: 1.770
> minwt;
12
```

We now find the codewords of LC having minimum weight:

```
> time wds := Words(LC, minwt);
Time: 151.860
> #wds;
42840
```

We now construct a design from the supports of the words *wds*. Note that codewords c and $2c$ have the same support, and thus contribute one block between them.

```
D := Design< 5, Length(LC) | wds >;
> D;
Design on 36 points with 21420 blocks

> BlockDegree(D);
12
> ReplicationNumber(D);
7140

> time G := AutomorphismGroup(D);
Time: 49.129
> G;
Permutation group G
acting on a set of cardinality 21456
Order = 9792 = 2^6 * 3^2 * 17
> time ChiefFactors(G);
    G
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  A(1, 17)               = L(2, 17)
    1
```

Chapter 27

Cryptosystems Based on Modular Arithmetic

27.1 Introduction

MAGMA contains most of the necessary tools to implement algorithms like RSA, Diffie-Hellman, Rabin's scheme, DSA, and similar cryptographic protocols based on modular arithmetic. We say 'most' of the tools since it currently does not implement any hash functions, but later versions will. MAGMA is highly optimised for modular arithmetic in terms of both algorithms and implementation. When primes are requested, MAGMA will actually prove that the numbers are prime using elliptic curve primality proving, unless the option `Proof := false` is used. In the latter case, the strong pseudo prime[1] probabilistic primality test is used with 20 random bases. The number of bases can be changed when using the `IsProbablePrime` function.

Cyrptographers will find MAGMA's efficient implementation of the following functions quite useful:

- Modular arithmetic including exponentiation, inverses, square roots, primitive roots
- Primality proving, probabilistic primality testing
- Random numbers, random primes
- Factorisation including trial division, Shanks square forms, Pollard rho, elliptic curve, mpqs
- Discrete logs including Pollard's method, baby-step giant-step, linear sieve, and Gaussian integer sieve.
- Legendre symbol, Jacobi symbol, Euler phi function, Carmichael lambda function
- Greatest common divisor, extended greatest common divisor algorithm, least common multiple
- Chinese remainder theorem

27.2 Example: Diffie-Hellman key exchange

The following example illustrates how easy it is to implement cryptographic protocols based on modular arithmetic within MAGMA. The example first selects a 160−bit prime q and then a 1024-bit prime p such that $p-1$ is divisible by q. We then create a base element g that has order q modulo p. This completes the parameter generation.

```
> q := RandomPrime (160: Proof := false);
> q;
```

[1] sometimes called the Miller-Rabin test

```
1337515134874490020173357215531009450590829772871
> Ilog2 (q);
159
> repeat
repeat>     got_it, p := RandomPrime (1024, 1, q, 1000 : Proof := false);
repeat> until got_it;
> p;
958085731357445446761306092342544057468757199232159208561376754132351855642406 3\
687099971497939650402010652318912664649002340468975126879797328794311274846885 8\
511218858302943917878233390934145245671375167605527243583846660070746646976863 7\
024368864000287009314028079649527237489139040773022882126340861653129 79
> Ilog2 (p);
1023
> tmp := Random(p);
> exp := (p - 1) div q;
> g := Modexp (tmp, exp, p);
> g;
909943466308741968513556146728669032553356505038976430440687981842909871008612 7\
505632383760336448155588201069259144303154280650830425342264397135461726101520 5\
952611003698228183612191569532019818110749495704832645085139838971885245155337 7\
719858827683982049536031668239277583842464264485868488587438685080807 69
> Modexp (g, q, p);
1
```

The `RandomPrime` function above is used in two different ways. The first time, we asked for a random prime which was up to 160−bits in length, and it returned the value q which happened to be exactly 160−bits (which is equivalent to $\lfloor \log_2 q \rfloor = 159$). The second time, we asked for a 1024−bit prime that was congruent to 1 modulo q (so that $p - 1$ would be divisible by q). The fourth argument, '1000', was the number of iterations to attempt finding this prime before giving up. Since there is a possibility of failure (for example, the user could possibly choose parameters for which no prime exists of the form being requested), using `RandomPrime` in this form returns two values: the first value is a boolean telling if it succeeded, and if it succeeded, the second value is the prime. We put the second `RandomPrime` call in a loop to be sure we found such a prime.

Now that the parameters are set up, we are able to do a pseudo-implementation of the protocol. First, Alice and Bob choose private keys and compute their corresponding public keys.

```
> A_priv := Random (p-1);
> A_pub := Modexp (g, A_priv, p);
> B_priv := Random (p-1);
> B_pub := Modexp (g, B_priv, p);
```

Alice and Bob now use each others public keys to compute the shared secret.

```
> A_sec := Modexp (B_pub, A_priv, p);
> B_sec := Modexp (A_pub, B_priv, p);
```

Finally, we check that their secrets are the same.

```
> A_sec eq B_sec;
```

```
true
> /* the secret is... */
> A_sec;
810941695296029024747717436052780061393060322180120076537102208019000039373644 5\
030715906030243949713228593807798082898694058235111880571153330772920256345465 2\
981538786479307989443422683355524973010379536158258827489366468380942188522772 1\
957530930588977688844563352831054745989644279050206844637462955623304870
```

27.3 Example: Hastad's broadcast attack on RSA

MAGMA has the tools to implement and experiment with several attacks on public key cryptosystems. Below is a simple example - Hastad's broadcast attack on RSA when proper padding is not used. In the simplest form of Hastad's attack, it is possible to recover a message M if it is encrypted and sent to three different people each having public exponent 3. The idea is to use the Chinese remainder theorem to reconstruct M^3 modulo the product of each person's modulus, and that value is the same as the integer M^3. Hence, M is retrieved by taking the cube root of the result.

The first step is to generate three 1024-bit RSA moduli that each allow public exponent 3. The `RSAModulus` command can be used to do this.

```
> N1 := RSAModulus (1024, 3);
> N1;
125388786274762932127690784395146627341474379367865898802277098621737750076328 3\
190477507897307532073908762693951348836877755797973797082488232805313718126640 9\
917794287669011126928672998402307425582249505229170356782594712253840590855142 1\
675390527731357051212792814457683608165818986174486455092581279890651793
> N2 := RSAModulus (1024, 3);
> N2;
112893011283983238269925524269925698047431688219083509708769536887763910049270 8\
973291332949714845143915604567460258129398606834484315709761820892800531853225 7\
619946085362173110066275071714432464303346178764350497815886575940881358574227 5\
148791782225209702335015030319402683201928555839431997857648498428842807
> N3 := RSAModulus (1024, 3);
> N3;
996023862062517253812776560181163255102675771849032024019389169108572062971769 6\
278514070361011210940098879219945257505547441961467071396069585596530583011105 2\
416758821648812187202846217451442347165822329983050982938121800258629117944184 6\
545551660128638881652659729971813746661983305339022267425083739689076 47
```

We will choose a random number between 1 and $\min(N1, N2, N3)$ as the message to encrypt.

```
> N1 lt N2;
false
> N2 lt N3;
false
> M := Random (N3);
> M;
344052923258672229282932886113656134673644260093483967715709390160516798595899 3\
385706012304230888983579582452347595654316082681078125417187760492879118616907 0\
```

```
362153719378035704686491271452585453605406734640448784019605587838979820940706 5\
809744395266101809073444765558770761261036326123441159510502215751286 23
```

Next, the message is encrypted for each of the three moduli.

```
> C1 := Modexp (M, 3, N1);
> C2 := Modexp (M, 3, N2);
> C3 := Modexp (M, 3, N3);
> C1;
996788336254746046024027234182638050221105018719436386399813855218984800568731 1\
863604504841808496514928509667291079413875491168135919150280872978311714993947 0\
771506617114838885891216272520061083505690006149536731447191884469295888764946 0\
443263710712034145561939465311317634141487240521317884691901922095963 58
> C2;
738266329732796048852392686111086847930952812434202227668696711290654105249872 3\
475481787388073612345990477429428362151842507318752270374354409080579986722601 6\
149771433012974645355459474633711995500392707003939473792379332760070350436820 1\
296791554355505450888466331059039954533011102336887322698673862425958 6
> C3;
870432025986436357394483675968432713665033281136857807215174345773815050453234 0\
084967576129788949764661643914055955288090250468934973667483933547740266783509 4\
581691796681483651760985501104571706446151232069867581886965757922996557404793 8\
330536735266995598017220437872300805759028352521838258238752281067205 79
```

The value of $C' = M^3$ can be computed using the Chinese remainder theorem. This is done below.

```
> ciphertexts := [C1, C2, C3];
> moduli := [N1, N2, N3];
> Cprime := CRT (ciphertexts, moduli);
> Cprime;
407263750708621323693403050874481444874800310989278120523313584708344105345271 9\
725312867043843495620178601290035126248683795357218422534469005646378445693714 8\
038826472974384779362569963835157070388688373303633641644176466092975615288314 1\
695088452759281666874425020997785876886688944410297947113906545946755630707751 4\
411659890538073092247443278318924794118048620148075973568178586526168258589728 2\
373564440656449333201459960201047576146904950705664088845846992410961428008980 7\
271964357412624092755904866781347559774418139410941725143072950096272300199583 0\
099711699543584265890113704167257645663407027517414392134992785210668366802494 2\
684871646096841810466415798364626124165693916460808175545969761535080256442665 1\
764515055557674492394508352732377417559019573722922634200045631831087007586281 4\
325635319538359046763499309743441056772020314665762158981049827301882448124424 6\
925550241949757973311329681134689567874662772504340367
```

Finally, we reconstruct the original message by taking the cube root.

```
> Iroot (Cprime, 3);
344052923258672229282932886113656134673644260093483967715709390160516798595899 3\
385706012304230888983579582452347595654316082681078125417187760492879118616907 0\
362153719378035704686491271452585453605406734640448784019605587838979820940706 5\
809744395266101809073444765558770761261036326123441159510502215751286 23
```

27.4 Primality proving

As mentioned earlier, MAGMA provides the option of proving primality, and in fact, it can even return a readble proof to you. The function `PrimalityCertificate` runs Morain's ECPP and returns a certificate, and then the function `PrimalityCertificate` can be used to convert that certificate into a readable proof. The option `Trust` indicates the point below which you are prepared to take primes on trust (i.e. not display a formal proof).

Let us illustrate with the value q that was used in section 27.2. Since the proof takes up a few pages, we only show part of it here.

```
> q := 133751513487449002017335721553100945059082977287 1;
> pc := PrimalityCertificate (q);
> IsPrimeCertificate (pc : ShowCertificate := true, Trust := 10^6);
Statement: N = 14200079 is prime
----------

We are going to use the following theorem:

Theorem 1:
  Let N - 1 =  m*p where p is an odd prime such that 2*p+1 > sqrt(N).
  If there exists an integer a such that
    1. a^((N - 1)/2) = -1 mod N, and
    2. a^(m/2) <> -1 mod N
  then N is prime.

Here:
  N - 1 = 106 * 133963
  It was already proved that p=133963 is a prime.
  2*p+1 = 267927 > sqrt(N) = 3768.29921848040087401695485 5

Let a := 11

Then:

  a^((N - 1)/2) = -1 mod N
  a^(m/2) = 3781767 mod N

Therefore, N=14200079 is prime.

Statement: N = 28400159 is prime
----------

We are going to use the following theorem:

Theorem 1:
  Let N - 1 =  m*p where p is an odd prime such that 2*p+1 > sqrt(N).
  If there exists an integer a such that
    1. a^((N - 1)/2) = -1 mod N, and
    2. a^(m/2) <> -1 mod N
  then N is prime.

Here:
```

```
  N - 1 = 2 * 14200079
  It was already proved that p=14200079 is a prime.
  2*p+1 = 28400159 > sqrt(N) = 5329.179955677984045909398984

Let a := 7

Then:

  a^((N - 1)/2) = -1 mod N
  a^(m/2) = 7 mod N

Therefore, N=28400159 is prime.
```

The proof continues on, the final lines being:

```
Statement: N = 995687686251595795850941705569519430795 3 is prime
----------

We are going to use the following theorem:

Theorem 4:
Let N be an integer greater than 1 and prime to 6. Let E be an
elliptic curve over Z/NZ, m and s two integers such that s divides
m. Suppose we have found a point P on E that satisfies mP = O_E, and
that for each prime factor q of s, we have verified that (m/q)P <> O_E.
If s > (sqrt[4](N)+1)^2, then N is prime.

Here:
  E : Y^2 = X^3 + 0 * X +  13537256947685289 .
  P =  (93283260018241286061018784047799194599 76,
        52295929049906383684286451539058987178 96, 1) .
  m * P = 55028159614939147721070141849703 * P = (0, 1, 0) .
  m := c*s where c = 1 and s = 55028159614939147721070141849703.
  It was already proved that q =  55028159614939147721070141849703 is a prime.
  (m/q)*P = (93283260018241286061018784047799194599 76,
52295929049906383684286451539058987178 96, 1) .
  s > (sqrt[4](N)+1)^2 = 99784151379379952272.45348058 .

Therefore, N is prime.

Statement: N = 13375151348744900201733572155310094505908297728 71 is prime
----------

We are going to use the following theorem:

Theorem 1:
  Let N - 1 =  m*p where p is an odd prime such that 2*p+1 > sqrt(N).
  If there exists an integer a such that
    1. a^((N - 1)/2) = -1 mod N, and
    2. a^(m/2) <> -1 mod N
  then N is prime.

Here:
```

```
  N - 1 = 134330790 * 995687686251595795850941705569519430795 3
  It was already proved that p=995687686251595795850941705569519430795 3 is a
  prime.
  2*p+1 = 199137537250319159170188341113903886159 07 > sqrt(N) =
1156509893980371453775462.441

Let a := 3

Then:

  a^((N - 1)/2) = -1 mod N
  a^(m/2) = 70263252405973033864663764392590404475171694483 4 mod N

Therefore, N=13375151348744900201733572155310094505908297728 71 is prime.

true
```

27.5 Integer factorisation

MAGMA's highly optimised factorisation routines can be quite useful in numerous applications. The examples below illustrates for three numbers. The timings were done on a 333 Mhz Sun Workstation.

```
> n := Random (2^100);
> n;
49826990526401287007564624954
> time Factorisation (n);
[ <2, 1>, <13450639, 1>, <1852216483038511665043, 1> ]
Time: 0.189
>
> n := Random (2^150);
> n;
74058637793741976936721008227802139460216154
> time Factorisation (n);
[ <2, 1>, <3, 1>, <11, 1>, <1149559, 1>, <140102799198689, 1>,
<6967126655838661037819, 1> ]
Time: 2.330
>
> n := Random (2^200);
> n;
48575727033873061443368896507947796979529231172281572477 3804
> time Factorisation (n);
[ <2, 2>, <516348517733, 1>, <46247313044493886301, 1>,
<508545572981008363362519 3947, 1> ]
Time: 45.149
```

Chapter 28

Elliptic Curve Cryptography

28.1 Introduction

Magma conatins an extensive library for working with elliptic curves over many different types of fields. Here we restrict our examples to the basic functions which can be used to implement elliptic curve cryptography (ECC). For a more in-depth discussion of Magma's elliptic curve functionality, please refer to the documents *Alebraic Geometry in Magma* and *An Overview of Magma*.

Useful functions discussed here include:

- Creation of elliptic curves over any finite field
- Point arithmetic
- Point counting (Schoof-Elkies-Atkin algorithm)
- Getting random points
- Order of a point
- Enumerate all points (for small fields)
- Discrete logarithms
- Supersingular testing
- Frobenius map and trace

28.2 Small example

This example is small enough that one can verify the computations by hand. In the first step, we create the elliptic curve $y^2 = x^3 + 16x + 11$ over GF(19) by specifying the two coefficients 16 and 11. We then list all the points, which are given in projective form. Next, we select two points (P and Q) and do some simple arithmetic operations.

```
> E := EllipticCurve ([K!16, K!11]);
> E;
Elliptic Curve defined by y^2 = x^3 + 16*x + 11 over GF(19)
> RationalPoints (E);
{ (11, 6, 1), (15, 4, 1), (6, 0, 1), (4, 5, 1), (8, 10, 1), (0, 7, 1), (5, 8,
1), (0, 1, 0), (17, 3, 1), (11, 13, 1), (0, 12, 1), (15, 15, 1), (17, 16, 1),
(8, 9, 1), (1, 3, 1), (4, 14, 1), (5, 11, 1), (1, 16, 1) }
```

```
> P := E![11, 6, 1];
> Q := E![15, 4, 1];
> P;
(11, 6, 1)
> Q;
(15, 4, 1)
> P + Q;
(17, 16, 1)
> P - Q;
(4, 5, 1)
> 2*P;
(4, 14, 1)
> -P;
(11, 13, 1)
```

Suppose we want to determine the order of P. Here is the naive method:

```
> i := 0;
> repeat
repeat> i := i + 1;
repeat> until (IsIdentity(i*P));
> i;
9
> 9*P;
(0, 1, 0)
```

And here is a much better method:

```
> Order(P);
9
```

28.3 Point counting

The use of randomly generated curves for ECC is often suggested, since some people fear that specially chosen curves may not be as secure. Unfortunately, the difficulty of efficient implementation of point counting algorithms prevent many people from having this option. However, MAGMA does quite well, as the following example illustrates. These computations were done on a 333 MHz Sun workstation.

```
> K<w> := GF(2^163);
> E := EllipticCurve ([K!1, Random(K), 0, 0, Random(K)]);
> E;
Elliptic Curve defined by y^2 + x*y = x^3 + (w^162 + w^159 + w^158 + w^153 +
    w^151 + w^150 + w^149 + w^148 + w^144 + w^141 + w^139 + w^137 + w^136 +
    w^134 + w^132 + w^131 + w^129 + w^127 + w^125 + w^124 + w^122 + w^120 +
    w^119 + w^118 + w^117 + w^115 + w^111 + w^110 + w^107 + w^106 + w^103 +
    w^100 + w^98 + w^97 + w^96 + w^95 + w^93 + w^92 + w^91 + w^90 + w^86 + w^85
    + w^84 + w^80 + w^79 + w^76 + w^74 + w^71 + w^66 + w^64 + w^58 + w^57 + w^54
```

```
    + w^52 + w^51 + w^49 + w^48 + w^46 + w^43 + w^41 + w^39 + w^37 + w^36 + w^34
    + w^31 + w^28 + w^25 + w^22 + w^19 + w^18 + w^17 + w^11 + w^9 + w^8 + w^7 +
    w^4 + w^3 + w)*x^2 + (w^162 + w^159 + w^158 + w^157 + w^156 + w^155 + w^154
    + w^152 + w^151 + w^150 + w^148 + w^146 + w^145 + w^143 + w^142 + w^140 +
    w^139 + w^132 + w^131 + w^130 + w^127 + w^125 + w^119 + w^117 + w^116 +
    w^114 + w^112 + w^111 + w^110 + w^108 + w^107 + w^106 + w^102 + w^101 +
    w^100 + w^95 + w^92 + w^91 + w^90 + w^87 + w^86 + w^81 + w^80 + w^79 + w^76
    + w^75 + w^73 + w^72 + w^71 + w^69 + w^68 + w^66 + w^61 + w^60 + w^57 + w^56
    + w^55 + w^54 + w^46 + w^43 + w^39 + w^38 + w^37 + w^36 + w^31 + w^29 + w^28
    + w^26 + w^23 + w^22 + w^19 + w^16 + w^15 + w^14 + w^12 + w^11 + w^10 + w^8
    + w^7 + w^6 + w^5 + w^2 + w) over GF(2^163)
> time Order(E);
1169201309864722334562947739523164912460682554893 2
Time: 43.810
```

Counting the points took just short of 44 seconds! To convince ourself that the result is correct, we select a random point and multiply it by the order:

```
> P := Random (E);
> P;
(w^161 + w^158 + w^156 + w^154 + w^153 + w^152 + w^151 + w^146 + w^142 + w^141 +
    w^139 + w^138 + w^137 + w^135 + w^134 + w^131 + w^129 + w^128 + w^124 +
    w^123 + w^122 + w^121 + w^119 + w^118 + w^117 + w^115 + w^114 + w^113 +
    w^112 + w^111 + w^109 + w^105 + w^104 + w^102 + w^100 + w^99 + w^97 + w^91 +
    w^90 + w^86 + w^84 + w^81 + w^79 + w^72 + w^69 + w^68 + w^66 + w^65 + w^64 +
    w^61 + w^59 + w^58 + w^57 + w^56 + w^55 + w^54 + w^52 + w^50 + w^48 + w^46 +
    w^44 + w^39 + w^37 + w^35 + w^33 + w^32 + w^30 + w^29 + w^26 + w^23 + w^21 +
    w^17 + w^16 + w^14 + w^11 + w^10 + w^8 + w^7 + w^2 + w + 1, w^161 + w^160 +
    w^157 + w^156 + w^155 + w^154 + w^153 + w^151 + w^149 + w^148 + w^147 +
    w^146 + w^144 + w^143 + w^141 + w^140 + w^139 + w^138 + w^136 + w^133 +
    w^130 + w^129 + w^126 + w^122 + w^121 + w^119 + w^118 + w^117 + w^116 +
    w^115 + w^113 + w^108 + w^107 + w^106 + w^105 + w^103 + w^101 + w^99 + w^97
    + w^94 + w^93 + w^91 + w^90 + w^87 + w^86 + w^85 + w^84 + w^79 + w^78 + w^77
    + w^76 + w^75 + w^74 + w^73 + w^72 + w^71 + w^68 + w^66 + w^64 + w^63 + w^62
    + w^59 + w^55 + w^54 + w^51 + w^50 + w^49 + w^48 + w^47 + w^45 + w^44 + w^41
    + w^40 + w^39 + w^37 + w^35 + w^34 + w^33 + w^29 + w^28 + w^27 + w^24 + w^23
    + w^22 + w^17 + w^16 + w^14 + w^13 + w^11 + w^9 + w^8 + w^7 + w^6 + w^5 +
    w^3 + w^2 + w + 1, 1)
> 1169201309864722334562947739523164912460682554893 2 * P;
(0, 1, 0)
```

28.4 Elliptic curve discrete logarithms

As any cryptographer knows, computing discrete logs on elliptic curves can be a very difficult task. However, if the curve was not chosen securely, the computation may be feasible. MAGMA contains a fast implementation of the Pohlig-Hellman and the van Oorschot-Wiener parallel collision search algorithms. In the example below, the largest prime factor of the order of the curve is 50-bits.

```
> K := GF(RandomPrime(97));
```

```
> E := EllipticCurve([Random(K), Random(K)]);
> E;
Elliptic Curve defined by y^2 = x^3 + 135188524256085569078553345*x +
360541340928873474295284 4829 over GF(771788380928187415852295 11529)
> FactoredOrder (E);
[ <2, 1>, <53, 1>, <753185801653, 1>, <966696722854727, 1> ]
> P := Random(E);
> FactoredOrder (P);
[ <2, 1>, <53, 1>, <753185801653, 1>, <966696722854727, 1> ]
> Q := Random(Order(E))*P;
> P;
(640524097383373850860862978 26, 589649768682150829208925106 20, 1)
> Q;
(463404265201609898207462545 12, 544211774995752360458947627 77, 1)
> time m := Log (P, Q);
Time: 642.929
> m;
428026159006183575324111826 69
> m*P - Q;
(0, 1, 0)
```

The computation took less than 11 minutes.

28.5 Example: ECDSA

Below is a complete example of an elliptic curve digital signature algorithm implementation.[1] First we generate an elliptic curve over a 160-bit prime field which has an order divisible by a large prime n of at least 150-bits. It doesn't take long before such a curve is found. We check that the curve is not supersingular because of the Menezes-Okamato-Vanstone attack on such curves.

```
> p := RandomPrime (160);
> K := GF(p);
> repeat
repeat> E := EllipticCurve ([Random(K), Random(K)]);
repeat> fo := FactoredOrder (E);
repeat> n := fo[#fo][1];
repeat> until Ilog2 (n) ge 150;
> Ilog2 (n);
151
> E;
Elliptic Curve defined by y^2 = x^3 + 8256373946658448558065657696224929825723 2\
1153916*x + 263724712391975536517759024869070389806362633529 over
GF(1154811871861712421275423566665495891399485108263)
> IsSupersingular(E);
false
> FactoredOrder (E);
[ <683, 1>, <16907933702221265318820261904575354107723439 17, 1> ]
```

[1] For information on ECDSA, see *Elliptic Curve DSA (ECDSA): An Enhanced DSA* by Don Johnson and Alfred Menezes, available at http://www.certicom.com/ecc/wecdsa.htm#elli

To complete the parameter generation, a base point of order n must be selected.

```
> n;
169079337022212653188202619045753541077234391 7
> P := Random(E) * (Order(E) div n);
> P;
(790453240443123790554571249483379206427271956600,
3258432896981432224044590682157161119702722270 44, 1)
> Order(P) eq n;
true
```

The private key d is chosen as a random integer mod n, and from that the public key Q is computed as d times the base point P.

```
> d := Random(n);
> Q := d*P;
> Q;
(92051992638592414839845871115281857412859677450 2,
5538657680040972147619173043288824060254680893 06, 1)
```

Let the message M be a random number modulo n.

```
> M := Random(n);
> M;
16546360347439998896191715086481682661814383 60
```

To sign M, we first select a random number k and compute the point $k*P$. The x-coordinate of the result is extracted, and that value modulo n is taken to be r. Note that this $x-$coordinate is a finite field element which must be coerced to an integer. Then s is computed to be $k^{-1}(M + dr)$ mod n and the pair (r, s) is the signature for M.

```
> k := Random(n);
> kp_seq := ElementToSequence (k*P);
> kp_seq;
[ 70000852887290387781319548418589617585030633 8609,
5822534211181860603691913019357953599940908765 03, 1 ]
> r := (IntegerRing()!kp_seq[1]) mod n;
> r;
20073600943493614036641336476515790555956971
> s := (Modinv (k, n) * (M + d*r)) mod n;
> s;
11079729199993684354928834560555698267526029 19
```

To verify the signature, we first check that r and s are in the range of $[1, n-1]$. If that passes, set u_1 to $M * s^{-1}$ mod n, u_2 to $r * s^{-1}$ mod n, and let v be the x-coordinate modulo n of the point $u_1 * P + u_2 * Q$. The signature is valid if and only if v equals r.

```
> r in [1..n-1];
```

```
true
> s in [1..n-1];
true
> w := Modinv (s, n);
> u1 := M*w mod n; u2 := r*w mod n;
> temp := u1*P + u2*Q;
> temp;
(70000852887290387781319548418589617585030633860 9,
582253421118186060369191301935795359994090876503, 1)
> temp_seq := ElementToSequence (temp);
> v := (IntegerRing()!temp_seq[1]) mod n;
> v;
200736009434936140366413364765157905559569 71
> v eq r;
true
```

28.6 Example: Scalar multiply using the Frobenius map

Several cryptographic papers have showed the speed advantages of specially chosen curves, such as anamolous binary curves. This section illustrates some of the MAGMA tools for experimenting with these ideas by implementing the algorithms from Jerome Solinas' Crypto '97 paper entitled "An Improved Algorithm for Arithmetic on a Family of Elliptic Curves"[2].

We start by using a curve from Solinas' paper of order equal to twice a prime r.

```
> q := 2;
> m := 163;
> K<z> := GF(q^m);
> Esub := EllipticCurve([GF(q)| 1, 1, 0, 0, 1]);
> E := BaseExtend(Esub, K);
> E;
Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 1 over GF(2^163)
> FactoredOrder (E);
[ <2, 1>, <5846006549323611672814741753598448348329118574063, 1> ]
> r := $1[2][1];
> Trace(Esub);
1
```

Since trace is 1, the Frobenius endomorphism ϕ satisfies $\phi^2 - \phi + 2 = 0$. Hence, this curve has complex multiplication by $\tau = (1 + \sqrt{-7})/2$. We create that element in the maximal order of the quadratic field $\mathbb{Q}(\sqrt{-7})$. When working in this order, elements are represented as a $\mathbf{Z}-$module over the basis $[1, (1+\sqrt{-7})/2]$. Since τ appears as one of these basis elements, our implementation will be slightly simplified.

```
> F<t> := QuadraticField (-7);
> t^2;
-7
```

[2] We are grateful to Yukio Tsuruoka for helping us understand this algorithm as well as his own related results. Due to space constraints, we were unable to include his algorithm in this document.

```
> R<tau> := MaximalOrder (F);
> F!tau;
1/2*(1 + t)
> tau^2 - tau + 2;
0
```

Next, we choose a point of order r on the curve, and select a random integer k to be used for the scalar multiply.

```
> P := (#E div r) * Random(E);
> P;
(z^160 + z^159 + z^158 + z^157 + z^156 + z^155 + z^154 + z^153 + z^151 + z^148 +
    z^147 + z^146 + z^144 + z^143 + z^142 + z^140 + z^135 + z^133 + z^131 +
    z^130 + z^129 + z^128 + z^127 + z^126 + z^125 + z^124 + z^122 + z^121 +
    z^114 + z^113 + z^112 + z^109 + z^107 + z^106 + z^102 + z^99 + z^96 + z^93 +
    z^91 + z^90 + z^89 + z^88 + z^86 + z^85 + z^84 + z^83 + z^82 + z^79 + z^78 +
    z^77 + z^76 + z^74 + z^72 + z^70 + z^67 + z^65 + z^64 + z^62 + z^59 + z^57 +
    z^55 + z^54 + z^53 + z^52 + z^51 + z^49 + z^48 + z^47 + z^46 + z^41 + z^40 +
    z^39 + z^37 + z^36 + z^35 + z^32 + z^31 + z^28 + z^24 + z^23 + z^20 + z^18 +
    z^17 + z^16 + z^15 + z^13 + z^12 + z^11 + z^10 + z^9 + z^7 + z^5 + z^4 + z^3
    + z^2, z^162 + z^159 + z^157 + z^156 + z^154 + z^149 + z^148 + z^146 + z^142
    + z^141 + z^136 + z^133 + z^132 + z^131 + z^130 + z^127 + z^126 + z^125 +
    z^122 + z^120 + z^119 + z^118 + z^117 + z^112 + z^110 + z^107 + z^106 +
    z^103 + z^98 + z^96 + z^95 + z^93 + z^91 + z^87 + z^86 + z^83 + z^79 + z^78
    + z^77 + z^76 + z^71 + z^70 + z^69 + z^65 + z^64 + z^63 + z^61 + z^59 + z^58
    + z^57 + z^55 + z^46 + z^45 + z^44 + z^39 + z^37 + z^36 + z^35 + z^34 + z^33
    + z^32 + z^31 + z^30 + z^27 + z^25 + z^24 + z^23 + z^20 + z^18 + z^16 + z^14
    + z^13 + z^12 + z^11 + z^6 + z^3 + z + 1, 1)
> k := Random (r);
> k;
473654998307453862947701619761191425467867975556 9
```

We now want to write the scalar k in terms of τ so that the scalar multiply can be done quickly. By using the NAF encoding method, the expected density (i.e. number of nonzero coefficients) is one third the encoding length. To make the encoding length small, reduction of k modulo $\tau^m - 1$ is performed and the result is represented as a linear combination of 1 and τ. Fortunately, we can do this computation with much less code in MAGMA than what was given in Solinas' paper, because MAGMA provides a mod function[3] for the maximal order R, and because τ is an element of the integral basis. The code below performs the reduction, and then obtains a τ-adic NAF encoding of the result.

```
> modulus := tau^m - 1;
> k_mod_modulus := k mod modulus;
> seq := Eltseq (k_mod_modulus);
> x := seq[1];
> y := seq[2];
> S := [];
> while x ne 0 or y ne 0 do
while> if IsOdd (x) then
```

[3]The latest version of MAGMA has a mod function for the maximal order of $\mathbb{Q}(\sqrt{d})$ where d is one of the following values: -1, -2, -3, -7, -11, 2, 3, 5, 13, 17

```
while|if>   u := 2 - ( (x-2*y) mod 4);
while|if> else
while|if>   u := 0;
while|if> end if;
while> x := x - u;
while> Append (~S, u);
while> ytmp := y;
while> y := - (x div 2);
while> x := ytmp - y;
while> end while;
```

Before doing the scalar multiply, let's check that the encoding actually worked:

```
> s := #S;
> sum := R!0;
> for i in [s..1 by -1] do
for>     sum := tau*sum + S[i];
for> end for;
> sum;
-99195612086343785322478 8 + 589253799488616101896173*tau
> k_mod_modulus;
-99195612086343785322478 8 + 589253799488616101896173*tau
> sum eq k_mod_modulus;
true
```

Finally, we do the scalar multiply and verify the answer is correct:

```
> phi := FrobeniusMap (E);
> R := Identity (E);
> for i in [s..1 by -1] do
for>     index := S[i];
for>     if index gt 0 then
for|if>         R := phi(R) + P;
for|if>     elif index lt 0 then
for|if>         R := phi(R) - P;
for|if>     else
for|if>         R := phi(R);
for|if>   end if;
for> end for;
> print "result: ", R;
result:  (z^156 + z^155 + z^153 + z^151 + z^148 + z^147 + z^146 + z^145 + z^141
    + z^140 + z^139 + z^138 + z^136 + z^134 + z^132 + z^131 + z^129 + z^125 +
    z^122 + z^119 + z^116 + z^115 + z^113 + z^112 + z^109 + z^108 + z^104 +
    z^103 + z^98 + z^97 + z^96 + z^94 + z^93 + z^91 + z^90 + z^86 + z^82 + z^80
    + z^79 + z^78 + z^77 + z^76 + z^73 + z^72 + z^71 + z^69 + z^67 + z^66 + z^62
    + z^57 + z^55 + z^52 + z^51 + z^49 + z^47 + z^44 + z^41 + z^40 + z^39 + z^37
    + z^35 + z^34 + z^33 + z^30 + z^27 + z^24 + z^19 + z^17 + z^16 + z^15 + z^13
    + z^11 + z^7 + z^6 + z^5 + z^3 + z^2 + 1, z^162 + z^160 + z^157 + z^156 +
    z^153 + z^151 + z^143 + z^141 + z^140 + z^137 + z^136 + z^131 + z^130 +
    z^128 + z^127 + z^126 + z^122 + z^121 + z^120 + z^119 + z^115 + z^114 +
    z^112 + z^108 + z^107 + z^105 + z^101 + z^98 + z^95 + z^93 + z^92 + z^88 +
```

```
    z^87 + z^86 + z^81 + z^79 + z^78 + z^74 + z^68 + z^67 + z^65 + z^63 + z^61 +
    z^57 + z^53 + z^52 + z^51 + z^50 + z^49 + z^48 + z^47 + z^45 + z^44 + z^43 +
    z^39 + z^38 + z^36 + z^31 + z^30 + z^27 + z^25 + z^16 + z^15 + z^12 + z^11 +
    z^9 + z^7 + z^4 + z^3 + z + 1, 1)
> kP := k*P;
> print "expected to get: ", kP;
expected to get:  (z^156 + z^155 + z^153 + z^151 + z^148 + z^147 + z^146 + z^145
    + z^141 + z^140 + z^139 + z^138 + z^136 + z^134 + z^132 + z^131 + z^129 +
    z^125 + z^122 + z^119 + z^116 + z^115 + z^113 + z^112 + z^109 + z^108 +
    z^104 + z^103 + z^98 + z^97 + z^96 + z^94 + z^93 + z^91 + z^90 + z^86 + z^82
    + z^80 + z^79 + z^78 + z^77 + z^76 + z^73 + z^72 + z^71 + z^69 + z^67 + z^66
    + z^62 + z^57 + z^55 + z^52 + z^51 + z^49 + z^47 + z^44 + z^41 + z^40 + z^39
    + z^37 + z^35 + z^34 + z^33 + z^30 + z^27 + z^24 + z^19 + z^17 + z^16 + z^15
    + z^13 + z^11 + z^7 + z^6 + z^5 + z^3 + z^2 + 1, z^162 + z^160 + z^157 +
    z^156 + z^153 + z^151 + z^143 + z^141 + z^140 + z^137 + z^136 + z^131 +
    z^130 + z^128 + z^127 + z^126 + z^122 + z^121 + z^120 + z^119 + z^115 +
    z^114 + z^112 + z^108 + z^107 + z^105 + z^101 + z^98 + z^95 + z^93 + z^92 +
    z^88 + z^87 + z^86 + z^81 + z^79 + z^78 + z^74 + z^68 + z^67 + z^65 + z^63 +
    z^61 + z^57 + z^53 + z^52 + z^51 + z^50 + z^49 + z^48 + z^47 + z^45 + z^44 +
    z^43 + z^39 + z^38 + z^36 + z^31 + z^30 + z^27 + z^25 + z^16 + z^15 + z^12 +
    z^11 + z^9 + z^7 + z^4 + z^3 + z + 1, 1)
> R eq kP;
true
```

Chapter 29

LLL and Lattice Based Ciphers

29.1 Introduction

The LLL lattice reduction algorithm has proven itself to be a very important tool in cryptography. MAGMA has a highly optimised implementation based on the FP-LLL algorithm of Schnorr and Euchner and the de Weger integral algorithm. For more details on MAGMA's lattice machinery, refer to the document *Module Theory in Magma* by Allan Steel.

In this chapter we demonstrate how effective MAGMA's LLL can be for cryptanalytic applications by implementing and cryptanalyzing the Merkle-Hellman knapsack cryptosystem.

29.2 Merkle-Hellman knapsack scheme

Our implementation of the Merkle-Hellman Knapsack scheme is based upon the description given in *The Rise and Fall of Knapsack Cryptosystems* by Andrew Odlyzko. Let n be the number of elements in the knapsack. Then we create a superincreasing knapsack $b_1, \ldots, b_n$ with $b_1 \approx 2^n$ and $b_n \approx 2^{2n}$. We then choose a value $M > \sum_{j=1}^{n} b_j$ and an integer W such that $\gcd(M, W) = 1$. Let π be a random permutation of the integers $1, \ldots, n$. The public weights are created by $a_j = b_{\pi(j)} W \bmod M$. The trapdoor private information is the b_j, M, W, and π. The MAGMA code below is one way of performing this parameter generation.

```
function KnapsackParamGeneration (n)

    max_increment := 2^n;
    /* generate  b_1 : the first element of the vector */
    b_i := 2^n + Random (1, max_increment);
    b_vec := [b_i];
    sum := b_i;
    for i in [2..n] do
        b_i := sum + Random (1, max_increment);
        sum +:= b_i;
        Append (~b_vec, b_i);
    end for;

    M := NextPrime (sum);
    W := Random (1, M);
```

```
    aprime_vec := [(b * W) mod M : b in b_vec];

    pi := Random(Sym(n));

    /* The public weights are... */
    a_vec := [ aprime_vec[Image(pi, i)] : i in [1..n] ];

    return a_vec, b_vec, M, W, pi;
end function;
```

To encrypt a stream of n bits, we simply add the a_j's together where the corresponding input bit is a 1.

```
function KnapsackEncrypt (bit_stream, a_vec)

    n := #bit_stream;
    if n ne #a_vec then
        print "ERROR: bit_stream must be same length as a_vec";
        return 0;
    end if;

    /* assume  bit_stream  was entered properly (contains only 1's and 0's) */
    return &+[a_vec[i]*bit_stream[i] : i in [1..n]];

end function;
```

Decryption is done with the trapdoor information. By multiplying the sum s by W^{-1} mod M, we are able to find the original bit stream using the superincreasing property of the b_j and the permutation π.

```
function KnapsackDecrypt (s, b_vec, M, W, pi)

    c := (s * Modinv (W, M)) mod M;

    n := #b_vec;
    pi_inv := pi^-1;
    bit_stream := [0 : i in [1..n]];

    for j in [n..1 by -1] do
       if c ge b_vec[j] then
           bit_stream[Image (pi_inv, j)] := 1;
           c -:= b_vec[j];
       end if;
    end for;

    return bit_stream;

end function;
```

Here is a small example, using a knapsack of length 50 and a randomly generated bit stream.

```
> n := 50;
> a_vec, b_vec, M, W, pi := KnapsackParamGeneration (n);
> a_vec;
[ 13869155476683245329960713132 67, 480859449341809389615925150721,
396831847319158859761380910988, 110049219337727190162580928 2566,
602163238313232210355415916295, 225743102342246789252109786635,
178027020140160005094859199390, 779220245900000096190424559705,
111914404605111713523853391 4655, 141868695108843287861598399 8936,
899299012420785534489933598742, 565071352129143828482957948937,
539578214665453201619177094 57, 313904821458716880518680856652,
519241475289568336576750859767, 128966534294150566836529839 9433,
238900603596419042941537807273, 536259692176238414925018681313,
933360455834373612992698096458, 774202548749424545086539440 08,
135724758580565710227498315 4542, 265062672662096748950202024392,
146354637187046061342168390 1684, 780541407523099261353546130279,
409719664966522686344206604523, 313884181857775522859495247905,
143544438447419232415795254 6261, 753326465780188459888065768459,
424115371126362588790544114246, 100015480854990513098732369 1901,
806748883690625446614140743 51, 328517760626549131196213524 33,
159142370990238549173704452 5729, 116457329512318160573886274 0531,
116686610632868364861593880 6544, 508385025612481068258268255595,
100932497848672487171084969 0549, 158835335282790671459813002 8095,
800259809134539255416029873707, 768702586293351853940869658594,
632007950574899716493836675 62, 798217142168267708536649714137,
105238156409833827597926481 2466, 994288605478880536515871422188,
289608215198805540201911271638, 179259316982728917034735283463,
821652935065312755697389840491, 220243707382439118104488004 24,
667360608572892937088022870 26, 145362079149720553476747758142 ]
> bit_stream := [Random(0,1): i in [1..n]];
> bit_stream;
[ 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0,
1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1 ]
> s := KnapsackEncrypt (bit_stream, a_vec);
> s;
125484249739539893575227733579 68
```

And the decryption:

```
> KnapsackDecrypt (s, b_vec, M, W, pi);
[ 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0,
1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1 ]
```

29.3 Cryptanalysis with LLL

LLL has aided cryptologists in breaking many of the variants of knapsack cryptosystems. Below is a simple MAGMA program written by Allan Steel which can be used to break the Merkle-Hellman knapsack scheme. The general idea of this attack is due to Schnorr and Euchner.

The code sets up a lattice X of the form:

$$b_1 \quad = (2, 0, \ldots, 0, na_1, 0)$$

$$\begin{aligned} b_2 &= (2, \ldots, 0, na_2, 0) \\ &\vdots \\ b_n &= (0, \ldots, 2, na_n, 0) \\ b_{n+1} &= (1, \ldots, 1, ns, 1). \end{aligned}$$

Then a vector $v = (v_1, \ldots, v_{n+2}) \in X$ such that the norm of v is $n + 1$ and

$$v_1, \ldots, v_n, v_{n+2} \in \{\pm 1\}, v_{n+1} = 0,$$

yields a solution $x_i = |v_i - v_{n+2}|/2$ for $i = 1, \ldots, n$ to the knapsack.

The code repeatedly applies LLL with parameter 0.999[1] until a solution is found. It is possible to use other reduction algorithms that MAGMA provides to possibly speed up the process, but we didn't find the need for these other tools for this application.

```
function SolveKnapsack(Q, s)
    n := #Q;
    X := RMatrixSpace(IntegerRing(), n + 1, n + 2) ! 0;
    for i := 1 to n do
        X[i][i] := 2;
        X[i][n + 1] := n * Q[i];
        X[n + 1][i] := 1;
    end for;
    X[n + 1][n + 1] := n * s;
    X[n + 1][n + 2] := 1;

    c := 0;
    while true do
        c +:= 1;
        "LLL number", c;
        X := LLL(X: Delta := 0.999);

        v := X[1];
        "Norm:", Norm(v);

        if forall{i: i in [1 .. n] cat [n + 2] | Abs(v[i]) eq 1} and
            v[n + 1] eq 0 then
            return [i: i in [1 .. n] | v[i] ne v[n + 2]];
        end if;
    end while;
end function;
```

Using this code, the knapsack was broken in less than a second:

```
> SolveKnapsack (a_vec, s);
LLL number 1
Norm: 51
[ 2, 6, 8, 9, 10, 11, 17, 19, 20, 22, 24, 25, 27, 29, 30, 31, 36, 40, 45, 46,
48, 49, 50 ]
> x := $1;
```

[1] the parameter must be in $(0.25, 1)$

```
> soln := [0 : i in [1..n]];
> for i in x do
for> soln[i] := 1;
for> end for;
> soln;
[ 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0,
1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1 ]
```

But this was an easy example! We used the same code to break a knapsack of length 100. It finished in 258 iterations and took only 2.5 minutes!

Chapter 30

McEliece Cryptosystem

30.1 Introduction

MAGMA also contains coding theory functions, including:

- Construction of general cyclic codes
- Construction of standard linear, cyclic, and alternant codes
- Combining codes: sum, intersection, (external) direct sum, Plotkin sum, concatenation
- Modifying a code: augment, extend, expurgate, lengthen, puncture, shorten, etc
- Changing the alphabet: extend field, restrict field, subfield subcode, trace code
- Construction of basic codes: Hamming, Reed-Muller, Golay, QR
- Construction of BCH codes and their generalizations: Alternant, BCH, Goppa, Reed-Solomon, generalized Reed-Solomon, Srivastava, generalized Srivastava
- Properties: cyclic, self-orthogonal, weakly self-orthogonal, perfect, etc .
- Minimum weight, words of minimum weight
- Weight distribution, weight enumerator, MacWilliams transform
- Complete weight enumerator, MacWilliams transform
- Coset weight distribution, covering radius, diameter
- Syndrome decoding, Alternant decoding
- Bounds: upper and lower bounds on the cardinality of a largest code having given length and minimum distance
- Automorphism groups of linear codes over GF(p) (prime p), GF(4)
- Testing pairs of codes for isomorphism over GF(p) (prime p), GF(4)

We demonstrate how these functions can be useful to the cryptographer by implementing the McEliece public key cryptosystem, and showing an attack on the system when certain precautions are not taken. The attack comes from Tom Berson's Crypto '97 paper *Failure of the McEliece Public-Key Cryptosystem Under Message-Resend and Related-Message Attack.*

Below is the MAGMA code for implementing this cryptosystem. This code was written by Damien Fisher, however some small style changes were made.

The first function generates a random Goppa code using the minimal polynomial of a random element from a degree 50 extension of GF(2^{10}). The generator matrix for this code will have 524 rows, 1024 columns, and will be able to correct up to 50 errors. These are the parameters originally suggested by McEliece.

```
function RandomCode()
    q := 2^10;
    K<w> := FiniteField(q);
    Kext := ext<K | 50: Optimize := false>;

    repeat
        g := MinimalPolynomial (Random(Kext),K);
    until Degree(g) eq 50;

    L := [x: x in K];

    return GoppaCode (L, g);
end function;
```

The function below creates the private and public keys using the `RandomCode` function. The private key consists of the tuple $< C, S, P >$ where C is the Goppa code, S is an invertible 524 by 524 scrambler matrix, and P is a 1024 by 1024 permutation matrix over GF(2). The public key G is a 524 by 1024 matrix over $GF(2)$.

```
function McElieceKey()
    // Magically pick C.
    C := RandomCode ();
    n := Length (C);
    k := Dimension (C);

    M := MatrixRing (GF(2), k);
    repeat S := Random(M); until IsUnit(S);

    perm := Random (Sym(n));

    G := S * GeneratorMatrix (C);
    I_n := MatrixRing (GF(2), n)!1;
    P := Parent (I_n) ! [I_n[j] ^ perm : j in [1 .. n]];
    G := G * P;

    return <C, S, P>, G;
end function;
```

Next, we give the functions to encrypt and decrypt. Encryption of a 1024-bit message m is done by multiplying m (treated as a vector over GF(2)) by G and then adding a random error vector e having t 1's in it. Decryption is done by first multiplying the ciphertext by P^{-1}, then decoding the Goppa code, and finally multiplying by S^{-1}.

```
function McElieceEncrypt (G, m, t)
    // Construct a random error vector e of hamming weight t.
    n := Ncols (G);
    V := VectorSpace (GF(2), n);
    e := Eltseq (Zero(V));

    count := 0;
    repeat
```

```
        pos := Random (1, n);
        if (e[pos] eq 0) then
            e[pos] := 1;
            count := count + 1;
        end if;
    until count eq t;

    return m * G + V!e;
end function;

function McElieceDecrypt (x, privKey)
    C := privKey[1];
    S := privKey[2];
    P := privKey[3];
    xp := x * (P^-1);
    succ, mS := Decode (privKey[1], xp);
    if not succ then
        print ("Could not decrypt x!");
        return mS;
    else
        return (InformationSpace(C) ! Coordinates (C, mS)) * S ^ -1;
    end if;
end function;
```

And now, we test it. This example uses an error of weight $t = 10$, which is not large enough for true cryptographic applications, but good enough for illustrative purposes.

```
> priv, pub := McElieceKey();
> m := Random (VectorSpace(GF(2), 524));
> m;
(0 1 0 0 0 1 1 1 0 1 0 1 0 0 1 1 1 0 0 0 0 1 1 0 1 0 1 0 0 0 0 1 0 0 0 1 1 0 0 1
    1 0 1 1 1 0 1 0 0 1 0 1 1 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0
    1 0 0 1 0 1 0 0 1 1 0 1 1 1 0 1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 1 1 1 0 1 0 1 1
    0 0 0 0 0 1 1 1 0 0 1 0 0 1 0 1 1 0 1 0 0 0 0 0 0 1 1 0 1 1 1 1 0 1 1 1 0 0
    1 1 0 0 1 1 1 1 1 1 1 0 1 0 1 1 0 1 0 0 1 0 1 0 0 0 0 0 0 1 0 0 1 1 0 1 1 0
    0 1 0 0 0 1 1 0 0 0 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0 1 1 0 1 1 0 0 1 0 1 1 0 1
    0 1 0 1 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1 0 1 1 0 0 0 0 0 0 0 0 1 0 0 1 0
    1 1 0 1 0 1 1 0 0 0 0 1 1 1 0 1 0 1 0 0 1 0 1 0 1 1 1 0 1 1 0 0 0 0 0 1 1 0
    0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 0 1 1 1 1 0 0 0 1 1
    1 0 0 1 0 1 0 0 0 1 1 1 1 0 1 1 1 0 1 0 1 0 1 0 1 1 0 1 1 0 0 0 0 0 0 1 0 0
    1 0 1 0 1 0 1 0 1 1 0 1 1 1 1 0 0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1
    1 1 0 0 0 0 1 0 1 0 0 0 1 0 1 1 0 1 0 1 0 1 1 1 0 0 0 1 0 1 0 1 1 1 0 0 0 0
    1 1 0 1 1 0 1 1 1 1 0 0 1 1 0 1 0 0 1 1 1 1 0 0 1 0 1 1 0 1 1 0 0 0 0 1 1 0
    0 0 1 1 0 0 0 0 0 1 1 1 0 1 0 0 1 0 0 0 0 0 0 1 0 1 1 1)
> c := McElieceEncrypt (pub, m, 10);
> c;
(0 0 1 1 0 1 1 0 1 0 0 0 0 1 1 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 0 0
    1 0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 1 1 1 1 0 1 1
    1 1 0 0 0 1 0 1 1 1 0 1 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0 1 1 0 1 1 1 0 1 1 1 1 0 0
    1 1 1 1 0 0 0 0 0 1 1 0 0 1 1 1 0 0 1 1 0 1 1 1 0 0 0 0 0 1 1 0 0 0 1 1 1 1 1 0
    0 0 0 0 1 0 0 0 0 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 0 0 1 1 0 1 0 0 0 0 1 0 1
    0 1 1 0 0 0 1 0 0 1 1 1 1 1 0 0 1 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0 1 1 1
    1 1 1 1 1 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 1 0 0 1 0 1 1 1 0 1 1 1 0 0 0 1 0 0 1
```

```
    1 1 1 1 1 0 0 0 0 0 1 0 1 1 0 1 1 1 0 1 1 0 0 1 1 1 1 1 1 0 0 1 0 0 0 0 1 0 1
    0 1 1 1 0 1 1 0 1 0 0 1 1 1 0 0 1 1 1 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 1 0 1 0
    0 1 1 0 0 1 0 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 1
    0 1 1 0 1 0 1 1 0 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0
    1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
    0 1 1 0 0 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 1 1 1
    1 0 0 0 1 1 1 0 1 1 1 0 0 0 1 1 0 0 1 1 0 0 0 1 1 1 0 0 0 0 1 1 1 0 0 0 1 0
    1 1 1 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 1 1 0 0 0 1 0 1 0 1 1 0 1 1 1 1 0 0
    0 0 0 1 0 0 0 0 0 1 1 0 0 0 1 0 0 1 0 1 1 0 1 1 0 0 0 0 1 1 1 1 0 1 1 1 0 1 0
    1 0 1 0 0 1 0 1 1 1 0 1 0 0 0 0 1 0 0 1 1 0 1 0 0 0 0 1 1 0 1 0 0 0 0 1 1 1
    0 1 0 1 1 1 0 0 1 1 0 1 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 1 0 1 0 1 0 0 0
    0 0 1 0 1 0 0 1 1 0 0 0 1 0 0 1 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 1 0 0 1 0 0 1
    1 1 0 1 1 1 1 1 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 1 1 1 0 1 0 1 1 0 1 1 1 1
    0 1 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 0 0 1 0 0 0 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0
    0 0 0 1 1 0 1 0 1 0 0 0 1 1 1 1 0 0 0 1 1 0 1 0 1 0 0 1 0 0 0 1 1 1 0 0 0 0
    0 1 0 0 1 1 1 0 0 1 1 1 1 1 0 0 0 0 1 0 1 1 1 0 0 0 0 1 1 1 1 1 0 1 1 1 1 1
    1 1 0 0 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 1 0 1 1 1 1 0 0 0 1 0
    1 0 0 1 0 1 1 0 1 1 0 1 0 1 1 0 1 0 0 0 0 1 1 1 0 1 0 1 0 0 1 1 0 0 1 1 1 1
    0 0 1 1 1 1 1 1 1 0 0 0 1 0 1 1 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 1
    0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 1 0 1 0 0 0 1 0 1 1 0 0 1 0 0 1 0 1 1)
> mprime := McElieceDecrypt (c, priv);
> mprime;
(0 1 0 0 0 1 1 1 0 1 0 1 0 0 1 1 1 0 0 0 0 1 1 0 1 0 1 0 0 0 0 1 0 0 0 1 1 0 0 1
    1 0 1 1 1 0 1 0 0 1 0 1 1 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0
    1 0 0 1 0 1 0 0 1 1 0 1 1 1 0 1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 1 1 1 0 1 0 1 1
    0 0 0 0 0 1 1 1 0 0 1 0 0 1 0 1 1 0 1 0 0 0 0 0 0 1 1 0 1 1 1 1 0 1 1 1 0 0
    1 1 0 0 1 1 1 1 1 1 1 0 1 0 1 1 0 1 0 0 1 0 1 0 0 0 0 0 0 1 0 0 1 1 0 1 1 0
    0 1 0 0 0 1 1 0 0 0 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0 1 1 0 1 1 0 0 1 0 1 1 0 1
    0 1 0 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1 0 1 1 0 0 0 0 0 0 0 0 1 0 0 1 0
    1 1 0 1 0 1 1 0 0 0 0 1 1 1 0 1 0 1 0 0 1 0 1 0 1 1 1 0 1 1 0 0 0 0 0 1 1 0
    0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 0 1 1 1 1 0 0 0 1 1
    1 0 0 1 0 1 0 0 0 1 1 1 1 0 1 1 1 0 1 0 1 0 1 0 1 1 0 1 1 0 0 0 0 0 0 1 0 0
    1 0 1 0 1 0 1 0 1 1 0 1 1 1 1 0 0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1
    1 1 0 0 0 0 1 0 1 0 0 0 1 0 1 1 0 1 0 1 0 1 1 1 0 0 0 1 0 1 0 1 1 1 0 0 0 0
    1 1 0 1 1 0 1 1 1 1 0 0 1 1 0 1 0 0 1 1 1 1 0 0 1 0 1 1 0 1 1 0 0 0 0 1 1 0
    0 0 1 1 0 0 0 0 0 1 1 1 0 1 0 0 1 0 0 0 0 0 0 1 0 1 1 1)
> m eq mprime;
true
```

30.2 Attacking McEliece using message resend

Following Tom Berson's article, we explain how to attack the McEliece system when not properly implemented. The general idea is that if we could guess 524 locations where the product $m \times G$ is not changed by the error vector e, then we can look at those corresponding 524 columns of G to get a relation that doesn't involve any error between the ciphertext and the plaintext. If these columns are linearly independent, then square matrix associated with them can be inverted, and consequently m can be retrieved.

It takes too much expected work to guess 524 correct columns if nothing is known about where the errors are. However, if the same message is encrypted twice using different random vectors, then the exclusive-or of the ciphertexts equals the exclusive-or of the error vectors, hence revealing

where most of the errors are. The only errors which are not revealed are those that occur in the same place for both encryptions. Berson showed that for $t = 50$, you expect to find around 95.1 of the 100 errors, and then the probability of correctly guessing 524 good values is very reasonable (i.e. 5 or 12 attempts is often enough).

Since in our example we are only allowing $t = 10$, we expect to find approximately 19.8 errors. In fact, most of the time we will find 20, and this simplifies our code slightly because we usually only need one attempt.

Here is our MAGMA code for attacking McEliece under message-resend.

```
procedure MessageResend()
    priv, pub := McElieceKey();
    n := 1024;
    k := 524;

    m := Random (VectorSpace(GF(2), k));
    print "original message is: ";
    print m;
    c1 := McElieceEncrypt (pub, m, 10);
    c2 := McElieceEncrypt (pub, m, 10);
    c1_plus_c2 := c1 + c2;
    print "Sum of two ciphertexts is:";
    print c1 + c2;

    pubT := Transpose (pub);
    repeat
        /*  create a new  k  by  k  matrix by selecting certain
            columns of  pub .  To simplify the coding, we actually
            work with rows of  Transpose (pub) . */
        new_mat_seq := [];
        rows_selected := [];
        i := 0;
        one := GF(2)!1;
        while i lt k do
            j := Random (1, n);
            if c1_plus_c2[j] eq one then
                /*  don't select a row corresponding to one of the errors */
                continue;
            end if;
            if j in rows_selected then
                /*  This row has already been selected.
                    Can't take it twice */
                continue;
            end if;
            /* row  j  is a keeper */
            Append (~rows_selected, j);
            for l in [1..k] do
                Append (~new_mat_seq, pubT[j][l]);
            end for;
            i +:= 1;
        end while;
        restricted_pub := Transpose (MatrixRing(GF(2), k)!new_mat_seq);
```

```
    until IsUnit (restricted_pub);

    /*  Let  c1'  ("c1prime")  be the corresponding columns of  c1 */
    c1prime_seq := [];
    for i in [1..k] do
        Append (~c1prime_seq, c1[rows_selected[i]]);
    end for;
    c1prime := VectorSpace (GF(2), k) ! c1prime_seq;

    m1prime := c1prime * restricted_pub^-1;

    if m ne m1prime then
        print "deciphering failed.";
    else
        print "deciphered message is: ";
        print m1prime;
    end if;

end procedure;
```

Here is a sample output:

```
> MessageResend();
original message is:
(0 0 1 1 0 0 1 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 0 0 1 0 0 1 0 1 1 1 1 0 0 1 1 1 1 0
    0 1 1 1 0 1 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 1 1 0 1 0 1 0 0 1 0 0 1 1 0 1 0 0
    1 1 1 0 1 0 0 1 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0
    0 1 0 0 0 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 0 1 1 0 1 1 1 0 0
    1 0 0 1 1 1 1 0 0 0 0 1 0 1 0 0 0 1 1 0 0 1 1 1 1 0 0 1 1 1 1 1 1 0 1 0 1 0
    0 0 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 0
    0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 1 1 1 0 0 0 1 0 0 1 1 1 0 1 0 1 0 1 1 0
    0 0 0 1 1 0 1 1 1 1 1 1 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0
    1 1 1 1 1 0 1 1 1 1 0 0 1 0 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 1 0
    1 0 1 0 1 1 1 0 1 0 1 1 0 1 1 0 0 0 0 1 0 0 1 1 0 1 1 1 0 0 1 1 1 0 0 1 0 0
    0 1 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 1 1 1 0 0 0
    0 1 1 0 1 1 0 1 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1 0 1 1 1 0 0 1 1 0 1
    0 0 0 0 1 0 1 1 0 1 0 1 1 0 1 0 1 0 1 0 0 0 1 0 1 0 0 1 1 0 0 1 1 1 0 1 1 0
    0 0 1 0 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 0 0 0 1 0)
Sum of two ciphertexts is:
(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 1 0 0 0 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 0 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 1 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 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 1 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 1 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 1 0 0 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 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 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 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 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 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 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 1 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 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 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 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 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 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 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 1 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 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 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 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 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 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 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)
deciphered message is:
(0 0 1 1 0 0 1 0 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 1 0 0 1 0 1 1 1 1 0 0 1 1 1 1 0
    0 1 1 1 0 1 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 1 1 0 1 0 1 0 0 1 0 0 1 1 0 1 0 0
    1 1 1 0 1 0 0 1 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0
    0 1 0 0 0 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 0 1 1 0 1 1 1 0 0
    1 0 0 1 1 1 1 0 0 0 0 1 0 1 0 0 0 1 1 0 0 1 1 1 1 0 0 1 1 1 1 1 1 0 1 0 1 0
    0 0 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 0
    0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 1 1 1 0 0 0 1 0 0 1 1 1 0 1 0 1 0 1 1 0
    0 0 0 1 1 0 1 1 1 1 1 1 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0
    1 1 1 1 1 0 1 1 1 1 0 0 1 0 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 1 0
    1 0 1 0 1 1 1 0 1 0 1 1 0 1 1 0 0 0 0 1 0 0 1 1 0 1 1 1 0 0 1 1 1 0 0 1 0 0
    0 1 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 1 1 1 0 0 0
    0 1 1 0 1 1 0 1 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1 0 1 1 1 0 0 1 1 0 1
    0 0 0 0 1 0 1 1 0 1 0 1 1 0 1 0 1 0 1 0 0 0 1 0 1 0 0 1 1 0 0 1 1 1 0 1 1 0
    0 0 1 0 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 0 0 0 1 0)
```

Notice that the sum of the two ciphertexts had exactly twenty 1's in it (as expected).

Chapter 31

Pseudo Random Bit Sequences

31.1 Introduction

MAGMA has some useful tools for analyzing and generating bit sequences, including:

- LFSR simulation
- Berlekamp-Massey algorithm
- Autocorrelation and Crosscorrelation functions
- Sequence decimation
- Shrinking generator
- Blum-Blum-Shub and RSA pseudo-random bit generators

Bits are represented by MAGMA as elements over GF(2). However not all of these functions are restricted to this finite field.

31.2 Linear Feedback Shift Registers

For a linear feedback shift register (LFSR) of length L, initial state $s_0, \ldots, s_{L-1} \in \mathrm{GF}(q)$, and connection polynomial $C(D) = 1 + c_1 D + c_2 D^2 + \ldots + c_L D^L$ (also over GF(q)), the j'th element of the sequence is computed as $s_j = -\sum_{i=1}^{L} c_i s_{j-i}$ for $j \geq L$. MAGMA includes the functions `LFSRStep` and `LFSRSequence` to simulate linear feedback shift registers. The former will advance a sequence one step to the next state of the LFSR. The latter will return a sequence of values that result from the output of the corresponding LFSR.

We illustrate with an example from *The Handbook of Applied Cryptography* on page 196. First a sequence of length 15 is generated using the primitive polynomial, and next we show the states of the LFSR as it advances the first three steps.

```
> K := GF(2);
> P<D> := PolynomialRing (K);
> P;
Univariate Polynomial Ring in D over Finite field of size 2
> C := 1 + D + D^4;
> S := [K| 0,1,1,0];
```

```
> LFSRSequence (C, S, 15);
[ 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1 ]
>
> /* advance the LFSR one step at a time */
> S := LFSRStep (C, S); S;
[ 1, 1, 0, 0 ]
> S := LFSRStep (C, S); S;
[ 1, 0, 0, 1 ]
> S := LFSRStep (C, S); S;
[ 0, 0, 1, 0 ]
```

Using these functions, it is possible to create your own functions to simulate stream ciphers like nonlinear combination generators, summation generators, nonlinear filter generators, etc.... Another common stream cipher is the shrinking generator, however this one is already programmed within MAGMA (command: `ShrinkingGenerator`).

31.3 Berlekamp-Massey algorithm

The Berlekamp-Massey algorithm can be used to compute the smallest LFSR connection polynomial that generates a sequence of finite field elements. In the example below, we start with 10 randomly chosen elements from the field GF(13^2), and compute a connection polynomial that generates that sequence. The `BerlekampMassey` function returns a minimal connection polynomial, and also the length L which tells the number of elements necessary to regenerate the sequence (this is necessary since the connection polynomial may be singular). Using this information, we regenerate the initial sequence using its first L elements.

```
> K<w> := GF(13^2);
> S := [Random(K) : i in [1..10]];
> S;
[ w^15, w^94, 11, w^136, w^100, w^86, w^148, 0, w^122, w^163 ]
> C<D>, L := BerlekampMassey (S);
> C;
w^50*D^5 + w^87*D^4 + w^131*D^3 + w^136*D^2 + w^64*D + 1
> L;
5
> T := S[1..L];
> T;
[ w^15, w^94, 11, w^136, w^100 ]
> LFSRSequence (C, T, #S);
[ w^15, w^94, 11, w^136, w^100, w^86, w^148, 0, w^122, w^163 ]
```

31.4 Example: Sequence decimation

Decimation of the sequence S by a value a value d means to create a new sequence by taking every d'th element of S with wrap-around such that the new sequence is the same length as the original. Given a primitive polynomial over GF(q), one can obtain another primitive polynomial by decimating an LFSR sequence obtained from the initial polynomial. This is demonstrated in the code below.

```
> K := GF(7);
> C<D> := PrimitivePolynomial(K, 2);
> C;
D^2 + 6*D + 3
```

In order to generate an LFSR sequence, we must first multiply this polynomial by a suitable constant so that the trailing coefficient becomes 1.

```
> C := C * Coefficient(C,0)^-1;
> C;
5*D^2 + 2*D + 1
```

We are now able to generate an LFSR sequence of length $7^2 - 1$. The initial state can be anything other than $[0, 0]$.

```
> S := LFSRSequence (C, [K| 1,1], 48);
> S;
[ 1, 1, 0, 2, 3, 5, 3, 4, 5, 5, 0, 3, 1, 4, 1, 6, 4, 4, 0, 1, 5, 6, 5, 2, 6, 6,
0, 5, 4, 2, 4, 3, 2, 2, 0, 4, 6, 3, 6, 1, 3, 3, 0, 6, 2, 1, 2, 5 ]
```

We decimate the sequence by a value d having the property $\gcd(d, 48) = 1$.

```
> T := Decimate(S, 1, 5);
> T;
[ 1, 5, 0, 6, 5, 6, 4, 4, 3, 1, 0, 4, 1, 4, 5, 5, 2, 3, 0, 5, 3, 5, 1, 1, 6, 2,
0, 1, 2, 1, 3, 3, 4, 6, 0, 3, 6, 3, 2, 2, 5, 4, 0, 2, 4, 2, 6, 6 ]
> B := BerlekampMassey(T);
> B;
3*D^2 + 5*D + 1
```

To get the corresponding primitive polynomial, we multiply by a constant to make it monic.

```
> B := B * Coefficient(B, 2)^-1;
> B;
D^2 + 4*D + 5
> IsPrimitive(B);
true
```

31.5 Other bit generators

In addition to the pseudo-random bit generators mentioned above, one can also use the Blum-Blum-Shub or RSA random bit generators. In the example below, we generate a 1024-bit Blum-Blum-Shub modulus N, and then count the number of 1's in a sequence of 1000 bits generated from the Blum-Blum-Shub algorithm using modulus N and a randomly chosen initial seed.

```
> N := BlumBlumShubModulus(1024);
> &+[IntegerRing()!b: b in BlumBlumShub (N, Random(N), 1000)];
489
```

Chapter 32

Finite Fields

32.1 Introduction

Several sections of this book give examples of MAGMA's functionality with finite fields. However, the topic is important enough in cryptography that it deserves its own chapter.

Some of the finite field functions that MAGMA provides include:

- Construction of GF(q) for any prime power, and arithmetic in that field
- Construction of towers of extensions
- Trace and norm of an element
- Order of an element
- Characteristic and minimum polynomials
- Square root (Tonelli-Shanks method for GF(p)), n-th root
- Construction of primitive and normal elements, testing for primitivity and normality
- Enumeration of irreducible polynomials
- Polhig-Hellman, baby-step giant-step, linear sieve, and Gaussian integer sieve discrete logarithm algorithms
- Factorisation of polynomials over finite fields
- Creating splitting fields
- Factorisation over splitting fields

32.2 Factorisation of polynomials over finite fields

The ability to efficiently factor polynomials over finite fields can be very handy. Below we give a simple example, where we find all fifth roots of an element modulo a prime p.

```
> p := RandomPrime (30);
> p;
1051175861
> Factorisation (p-1);
[ <2, 2>, <5, 1>, <7, 1>, <59, 1>, <127261, 1> ]
> K := GF(p);
```

```
> m := Random(K) ^ 5;
> m;
881494904
> /* we know  m  is a fifth power.  Now find all of its fifth roots */
> P<x> := PolynomialRing (K);
> f := x^5 - m;
> Factorisation (f);
[
    <x + 98800491, 1>,
    <x + 180305718, 1>,
    <x + 351879082, 1>,
    <x + 469153538, 1>,
    <x + 1002212893, 1>
]
> fact := $1;
> for i := 1 to 5 do
for> z := -Evaluate (fact[i][1], 0);
for> print z, "^5 -", m, "=", z^5 - m;
for> end for;
952375370 ^5 - 881494904 = 0
870870143 ^5 - 881494904 = 0
699296779 ^5 - 881494904 = 0
582022323 ^5 - 881494904 = 0
48962968 ^5 - 881494904 = 0
```

32.3 Factorisation over splitting fields

In the next example, we create a random degree 5 monic polynomial over GF(2^{20}). It factors into a linear factor and a fourth degree polynomial. We verify that factorisation is correct, and then factor it over the splitting field.

```
> K<w> := GF(2^20);
> P<X> := PolynomialRing (K);
> f := P![Random(K), Random(K), Random(K), Random(K), Random(K), 1];
> f;
X^5 + w^516299*X^4 + w^885970*X^3 + w^628673*X^2 + w^453256*X + w^639090
> Factorisation (f);
[
    <X + w^371300, 1>,
    <X^4 + w^21416*X^3 + w^726649*X^2 + w^806675*X + w^267790, 1>
]
> fact := $1;
> prod := fact[1][1] * fact[2][1];
> prod;
X^5 + w^516299*X^4 + w^885970*X^3 + w^628673*X^2 + w^453256*X + w^639090
> J<z> := SplittingField (f);
> Q<Y> := PolynomialRing(J);
> FactorisationOverSplittingField (f);
[
    <Y + z^78 + z^76 + z^75 + z^73 + z^72 + z^70 + z^66 + z^65 + z^63 + z^62 +
        z^60 + z^59 + z^56 + z^55 + z^54 + z^52 + z^48 + z^47 + z^43 + z^42 +
```

```
        z^39 + z^38 + z^37 + z^35 + z^34 + z^30 + z^24 + z^22 + z^21 + z^19 +
        z^9 + z^8 + z^7 + z^2 + z + 1, 1>,
    <Y + z^78 + z^77 + z^76 + z^75 + z^74 + z^71 + z^70 + z^69 + z^68 + z^63 +
        z^60 + z^58 + z^55 + z^52 + z^41 + z^39 + z^37 + z^31 + z^29 + z^28 +
        z^26 + z^24 + z^22 + z^21 + z^20 + z^19 + z^17 + z^16 + z^15 + z^13 +
        z^11 + z^10 + z^8 + z^6 + z^5 + z^4 + z^3 + z, 1>,
    <Y + z^79 + z^74 + z^73 + z^72 + z^71 + z^70 + z^68 + z^67 + z^66 + z^65 +
        z^64 + z^62 + z^59 + z^58 + z^57 + z^53 + z^51 + z^50 + z^48 + z^47 +
        z^43 + z^41 + z^40 + z^39 + z^38 + z^36 + z^33 + z^32 + z^31 + z^29 +
        z^27 + z^26 + z^23 + z^18 + z^17 + z^15 + z^14 + z^9 + z^6 + z^5 + z^3 +
        z + 1, 1>,
    <Y + z^79 + z^78 + z^75 + z^74 + z^73 + z^72 + z^71 + z^70 + z^69 + z^65 +
        z^63 + z^53 + z^52 + z^51 + z^50 + z^48 + z^47 + z^45 + z^42 + z^41 +
        z^40 + z^37 + z^35 + z^34 + z^33 + z^32 + z^31 + z^22 + z^21 + z^20 +
        z^16 + z^15 + z^13 + z^12 + z^10 + z^9 + z^8 + z^5 + z^4 + z^3 + z^2 +
        1, 1>,
    <Y + z^79 + z^74 + z^73 + z^72 + z^71 + z^70 + z^69 + z^65 + z^64 + z^63 +
        z^60 + z^58 + z^57 + z^55 + z^54 + z^50 + z^49 + z^48 + z^46 + z^45 +
        z^43 + z^41 + z^39 + z^38 + z^37 + z^34 + z^32 + z^31 + z^29 + z^28 +
        z^27 + z^26 + z^24 + z^23 + z^20 + z^18 + z^15 + z^14 + z^13 + z^9 + z^7
        + z^5 + z^4, 1>
]
Finite field of size 2^80
> fact2 := $1;
> prod := fact2[1][1];
> for i := 2 to 5 do
for> prod := prod * fact2[i][1];
for> end for;
> prod;
Y^5 + (z^79 + z^78 + z^77 + z^75 + z^70 + z^69 + z^67 + z^60 + z^58 + z^56 +
    z^55 + z^52 + z^50 + z^49 + z^47 + z^46 + z^43 + z^38 + z^36 + z^34 + z^32 +
    z^30 + z^29 + z^26 + z^24 + z^22 + z^21 + z^20 + z^13 + z^12 + z^11 + z^8 +
    z^4 + z^3 + z + 1)*Y^4 + (z^79 + z^78 + z^77 + z^73 + z^69 + z^68 + z^67 +
    z^66 + z^65 + z^62 + z^61 + z^58 + z^57 + z^56 + z^54 + z^53 + z^51 + z^49 +
    z^47 + z^44 + z^42 + z^41 + z^40 + z^39 + z^38 + z^37 + z^34 + z^33 + z^32 +
    z^31 + z^29 + z^26 + z^25 + z^21 + z^20 + z^19 + z^15 + z^14 + z^13 + z^11 +
    z^10 + z^9 + z^4 + z^3 + z)*Y^3 + (z^78 + z^76 + z^74 + z^73 + z^69 + z^68 +
    z^67 + z^59 + z^58 + z^57 + z^52 + z^51 + z^48 + z^44 + z^42 + z^35 + z^34 +
    z^31 + z^30 + z^29 + z^27 + z^26 + z^25 + z^23 + z^22 + z^21 + z^17 + z^16 +
    z^15 + z^14 + z^10 + z^9 + z^8 + z^5 + z^2)*Y^2 + (z^79 + z^78 + z^76 + z^75
    + z^74 + z^70 + z^69 + z^68 + z^66 + z^65 + z^63 + z^62 + z^61 + z^60 + z^55
    + z^52 + z^48 + z^44 + z^42 + z^41 + z^40 + z^39 + z^37 + z^36 + z^32 + z^30
    + z^26 + z^25 + z^24 + z^23 + z^21 + z^20 + z^17 + z^15 + z^14 + z^13 + z^8
    + z)*Y + z^79 + z^78 + z^77 + z^76 + z^73 + z^70 + z^67 + z^66 + z^65 + z^63
    + z^62 + z^61 + z^58 + z^56 + z^53 + z^50 + z^45 + z^43 + z^41 + z^39 + z^37
    + z^35 + z^34 + z^29 + z^27 + z^21 + z^19 + z^16 + z^12 + z^11 + z^9 + z^8 +
    z^5 + z^2 + 1
```

The result was given as a polynomial over GF(2). To get it as a polynomial over K, we can do the following:

```
> P!prod;
X^5 + w^516299*X^4 + w^885970*X^3 + w^628673*X^2 + w^453256*X + w^639090
```

Remark: you may have noticed that the fields GF(2^{20}) and GF(2^{80}) are represented differently within MAGMA. For GF(2^{20}), the element w is a generator of the multiplicative group. MAGMA is able to use such a representation for small fields because the factorisation of the order of the multiplicative group can quickly be obtained, and therefore a multiplicative generator can be found quickly. For larger fields, MAGMA does not attempt this computation.

32.4 Discrete logarithms

Our next example shows MAGMA's ability to compute discrete logarithms efficiently in prime fields. We generate an 100-bit prime p, and notice that the factorisation of $p-1$ would make it nearly impossible to perform a Pohlig-Hellman discrete logarithm because the largest prime factor is 27-digits. Thus, MAGMA internally uses an index calculus discrete logarithm algorithm, which will either be the Gaussian integer sieve or the linear sieve, both due to Coppersmith, Odlyzko, and Schroeppel. In most cases, MAGMA will choose the Gaussian integer sieve, since it is more efficient in practice.

```
> p := RandomPrime (100);
> p;
260162318040694245187833585521
> Factorisation (p-1);
[ <2, 4>, <5, 1>, <31, 1>, <1049041605002799375757393 49, 1> ]
> K := GF(p);
> b := Random (K);
> b;
170391440692636176963122157492
> t := b^Random(p-1);
> t;
211365012871333266382728822093
> time m := Log (b, t);
Time: 16.820
> m;
848871057208047326757796266 14
> b^m - t;
0
```

The logarithm took less than 17 seconds! But subsequent logarithms will in fact go much faster, since certain data does not need to be recomputed. For example, on the second call to the `Log` function, the result is computed in about 1 second:

```
> b := Random (K);
> b;
259508645995390351289990720032
> t := b^Random(p-1);
> t;
191771411850218210970256763443
> time m := Log (b, t);
Time: 1.170
> m;
154135004837539992186544114998
```

```
> b^m - t;
0
```

32.5 Implementation of Shamir's threshold scheme

In Shamir's t out of n threshold scheme, a set of n people each hold shares of a secret S, and any t of them can get together to compute S. However, less than t people can get no information about it.

The scheme is based upon polynomial interpolation in a finite field. Let K be a finite field. Initially we choose the secret randomly from K along with $t-1$ other random values from K. These values are used to create a polynomial f of degree $t-1$ over K such that $f(0) = S$. Below we do this for $t = 3$ and $n = 5$.

```
> t := 3;
> n := 5;
> p := RandomPrime (160);
> p;
831683565424322358470119661658063377313329498701
> Ilog2(p);
159
> K := GF(p);
> S := Random (K);
> S;
388005396389914678062277072148817245612709240026
> a_vec := [S];
> for i in [1..t-1] do
for>     Append (~a_vec, Random(K));
for> end for;
> P<x> := PolynomialRing(K);
> f := P!a_vec;
> f;
57081094028156195679730394773594513470795347883 3*x^2 +
    339162245841631570546880748441891441159716340186*x +
    388005396389914678062277072148817245612709240026
```

The next step is to compute shares of the secret. For persons $i \in [1..n]$, we let their share be $f(i)$.

```
> shares := [];
> for i in [1..n] do
for>     shares[i] := Evaluate (f, i);
for> end for;
> shares;
[ 466295017088785846936342106668590444167049560344,
228393875021362124647757133441271575106378409 26,
721005638478610491587817215491554140270133079174,
657430737452416088951076281366812605055467789 85,
552102389574996639797005936253698650156867436462 ]
```

Reconstructing the secret is done by polynomial interpolation. First, we choose 3 people at random to reconstruct S.

```
> p := [];
> s := [];
> for i in [1..t] do
for> repeat
for|repeat> person := K!Random (1, n);
for|repeat>  until person notin p;
for> Append (~p, person);
for> /* store this person's share */
for> Append (~s, shares[IntegerRing()!person]);
for> end for;
>
> p;
[ 1, 3, 4 ]
> s;
[ 46629501708878584693634210666859044416704956034 4,
72100563847861049158781721549155414027013307917 4,
65743073745241608895107628136681260505546778985 ]
```

Using the MAGMA command `Interpolation`, we retrieve the secret:

```
> Interpolation (p, s);
57081094028156195679730394773594513470795347883 3*x^2 +
    33916224584163157054688074844189144115971634018 6*x +
    38800539638991467806227707214881724561270924002 6
> g := $1;
> Evaluate (g, 0);
38800539638991467806227707214881724561270924002 6
> $1 eq S;
true
```

Chapter 33

Miscellaneous

33.1 NTRU

The NTRU cryptosystem is describe on the web site http://www.ntru.com/tutorials/pkcstutorial.htm. Here we illustrate how to program it in MAGMA, and verify the example from the web site is correct.

Arithmetic will be done in $R = \mathbf{Z}[X]$ with reduction modulo $X^N - 1$ for $N = 11$. We will also need to reduce modulo $p = 3$ and $q = 32$. Thus, we create rings $Rp = \mathbf{Z}[X]/p$ and $Rq = \mathbf{Z}[X]/q$.

```
> N := 11;
> q := 32;
> p := 3;
> R<X> := PolynomialRing( Integers() );
> Rp := PolynomialRing( GF(p) );
> Rq := PolynomialRing( ResidueClassRing( q ) );
```

We will need to compute inverses in the ring of truncated polynomials modulo p and q. Algorithms for doing this are given at http://www.ntru.com/tech.technical.htm. Below is an implementation of the algorithms within MAGMA:

```
function InverseModPrime( R, a, p, N )

    X := R.1;
    Rp := PolynomialRing( GF(p) );

    k := 0; b := R!1; c := R!0;
    f := a; g := X^N - 1;

    while true do
        f0 := Evaluate(f, 0);
        while f0 eq 0 do
            f := f div X; c := c * X; k +:= 1;
            f0 := Evaluate(f, 0);
        end while;

        if Degree(f) eq 0 then
            b := R!Rp!(b * Modinv( f0, p ));
```

```
            ans := b*X^(-k mod N) mod (X^N-1);
            break;
        end if;

        if Degree( f ) lt Degree( g ) then
            t := f; f := g; g := t;
            t := b; b := c; c := t;
        end if;

        f0 := Evaluate(f, 0);
        g0 := Evaluate(g, 0);
        u := (Modinv( g0, p ) * f0) mod p;
        f := R!Rp!(f - u*g);
        b := R!Rp!(b - u*c);
    end while;

    return ans;
end function;


function InverseModPrimePower( R, a, prime_power, N )

    ok, p, r := IsPrimePower( prime_power );
    if ok eq false then
        print "ERROR: function requires prime power";
        return 0;
    end if;

    b := InverseModPrime( R, a, p, N );
    q := p;
    X := R.1;
    while q lt prime_power do
        q := q*p;
        Rq := PolynomialRing( ResidueClassRing( q ) );
        b := R!Rq!( (b * (2 - a * b)) mod (X^N - 1) );
    end while;

    return b;
end function;
```

We take the example polynomials $f(X)$ and $g(X)$ from the web site, and compute

- Fp - the inverse of f in the ring $\mathbf{Z}[X]/(p, X^N - 1)$
- Fq - the inverse of f in the ring $\mathbf{Z}[X]/(q, X^N - 1)$.

The results are coerced back to polynomials in $\mathbf{Z}[X]$. It is easy to check that the inverses are correct. The values of Fp, p and $g(X)$ are used to compute the public key $h(X)$.

```
> f := -1+X+X^2-X^4+X^6+X^9-X^10;
> g := -1+X^2+X^3+X^5-X^8-X^10;
> Fp := InverseModPrimePower( R, f, p, N );
```

```
> Fq := InverseModPrimePower( R, f, q, N );
> Fp;
2*X^9 + X^8 + 2*X^7 + X^5 + 2*X^4 + 2*X^3 + 2*X + 1
> Fq;
30*X^10 + 18*X^9 + 20*X^8 + 22*X^7 + 16*X^6 + 15*X^5 + 4*X^4 + 16*X^3 + 6*X^2 +
    9*X + 5
> R!Rp!(f*Fp mod (X^N - 1));
1
> R!Rq!(f*Fq mod (X^N - 1));
1
> h := R!Rq!(p*Fq*g mod (X^N - 1));
> h;
16*X^10 + 19*X^9 + 12*X^8 + 19*X^7 + 15*X^6 + 24*X^5 + 12*X^4 + 20*X^3 + 22*X^2
    + 25*X + 8
```

To encrypt a message, we represent it as a polynomial having coefficients between $-p/2$ and $+p/2$, and randomly select a polynomial $r(X)$ having a fixed number of +1 and -1 coefficients. The encryption of $m(X)$ is $e(X) = r(X) * h(X) + m(X) \bmod \mathbf{Z}[X]/(q, X^N - 1)$. We use the polynomials $m(X)$ and $r(X)$ from the example on the NTRU web site.

```
> m := -1+X^3-X^4-X^8+X^9+X^10;
> r := -1+X^2+X^3+X^4-X^5-X^7;
> e := R!Rq!((r*h + m) mod (X^N - 1));
> e;
19*X^10 + 6*X^9 + 25*X^8 + 7*X^7 + 30*X^6 + 16*X^5 + 14*X^4 + 24*X^3 + 26*X^2 +
    11*X + 14
```

Decryption will require adjusting polynomials so that the coefficients are between $-p/2$ and $+p/2$ or $-q/2$ and $+q/2$. The following function performs the adjustment.

```
function AdjustPolynomial( R, a, q )

    coeffs := Eltseq( a );
    qdiv2 := q div 2;
    for i in [1..#coeffs] do
        if coeffs[i] gt qdiv2 then
            coeffs[i] -:= q;
        end if;
    end for;

    return R!coeffs;
end function;
```

Finally we can perform the decryption steps and verify that the solution is correct.

```
> a := R!Rq!((f*e) mod (X^N - 1));
> a;
25*X^10 + 29*X^9 + 5*X^8 + 7*X^7 + 6*X^6 + 7*X^5 + 10*X^4 + 21*X^3 + 22*X^2 +
    25*X + 3
> a := AdjustPolynomial( R, a, q );
```

```
> a;
-7*X^10 - 3*X^9 + 5*X^8 + 7*X^7 + 6*X^6 + 7*X^5 + 10*X^4 - 11*X^3 - 10*X^2 - 7*X
    + 3
> b := AdjustPolynomial( R, R!Rp!a, p);
> c := AdjustPolynomial( R, R!Rp!(Fp*b mod (X^N - 1)), p);
> c;
X^10 + X^9 - X^8 - X^4 + X^3 - 1
> c eq m;
true
```

33.2 Rijndael's linear diffusion matrix

In the paper "New Observations on Rijndael" by Sean Murphy and Matt Robshaw, several unusual properties of the block cipher Rijndael (Advanced Encryption Standard candidate finalist) are identified. In particular, they show that the constant c which is added on to each byte can be relocated into the key schedule, and then the linear diffusion layer that remains can be represented by a 128 by 128 bit matrix. A very surprising property of this matrix is that it has exponent 16. In other words, any 128-bit input vector is mapped to itself after at most 16 iterations of the linear diffusion layer.

Below we outline MAGMA computations which verify the correctness of their observations. To do this, we create matrices which correspond to ByteSub, ShiftRow, and MixColumn transformations of Rijndael. Upon multiplying the three matrices together, we get the same linear diffusion matrix that Murphy and Robshaw obtained. We finish by computing the minimal polynomial and verifying that it is a divisor of $x^{16} + 1$ in $\mathrm{GF}(2)[x]$.

When building the matrices corresponding to the three transformations, it will sometimes be convenient for us to first work at the byte level, and then using the byte level matrices to build the corresponding bit level matrices. An input byte matrix of the form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

is represented as the transpose of the vector

$$[a_{11}, a_{21}, a_{31}, a_{41}, a_{12}, a_{22}, a_{32}, a_{42}, a_{13}, a_{23}, a_{33}, a_{43}, a_{14}, a_{24}, a_{34}, a_{44}] .$$

At the bit level, we expand the above vector so that each byte is replaced by 8 bits with the most significant bits coming first.

Let us begin with the ByteSub transform. It is straightfoward to create the 8 by 8 matrix which performs the byte substitution. That matrix is given in the Rijndael specifications.

```
> seq := [
>  1,1,1,1,1,0,0,0,
>  0,1,1,1,1,1,0,0,
>  0,0,1,1,1,1,1,0,
>  0,0,0,1,1,1,1,1,
>  1,0,0,0,1,1,1,1,
>  1,1,0,0,0,1,1,1,
```

```
>  1,1,1,0,0,0,1,1,
>  1,1,1,1,0,0,0,1
>  ];
> L := Matrix(GF(2),8,8,seq);
```

To create the 128 by 128 bit matrix which acts as ByteSub on all 16 bytes, we can simply do:

```
> linear_diffusion_layer := Matrix(GF(2),128,128,[0: i in [1..128^2]]);
> for j in [1..128 by 8] do
>     InsertBlock(~linear_diffusion_layer, L, j, j);
> end for;
```

We build the matrix corresponding to the ShiftRow transformation first at the byte level: the matrix M will send the vector

$$[a_{11}, a_{21}, a_{31}, a_{41}, a_{12}, a_{22}, a_{32}, a_{42}, a_{13}, a_{23}, a_{33}, a_{43}, a_{14}, a_{24}, a_{34}, a_{44}]$$

to

$$[a_{11}, a_{22}, a_{33}, a_{44}, a_{12}, a_{23}, a_{34}, a_{41}, a_{13}, a_{24}, a_{31}, a_{42}, a_{14}, a_{21}, a_{32}, a_{43}]\,.$$

There are of course many ways to build this matrix within MAGMA. We choose to do it by swapping rows of the identity matrix.

```
> M := IdentityMatrix(GF(2), 16);
> SwapRows( ~M, 2, 6 );
> SwapRows( ~M, 3, 11);
> SwapRows( ~M, 4, 16);
> SwapRows( ~M, 6, 10 );
> SwapRows( ~M, 7, 15 );
> SwapRows( ~M, 8, 16 );
> SwapRows( ~M, 14, 10 );
> SwapRows( ~M, 12, 16 );
> M;
[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 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 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 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 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 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 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 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]
```

Creating the corresponding bit level matrix can be done by expanding this matrix out and inserting the 8 by 8 identity wherever the 1's are.

```
> i8 := IdentityMatrix(GF(2), 8);
> big_M := Matrix(GF(2),128,128,[0: i in [1..128^2]]);
> for i in [1..16] do
>     for j in [1..16] do
>         if M[i][j] eq 1 then
>             InsertBlock(~big_M, i8, (i-1)*8+1, (j-1)*8+1);
>         end if;
>     end for;
> end for;
```

We now form the matrix which corresponds to first doing ByteSub and then ShiftRow:

```
> linear_diffusion_layer := big_M*linear_diffusion_layer;
```

The final step is the MixColumn transformation (refered to the matrix D in the Murphy/Robshaw paper). MixColumn is a function which treats 4 bytes of data (one column of data of the state) as an element of $\mathrm{GF}(2^8)[x]$, and multiplies this element by a fixed $\mathrm{GF}(2^8)$ polynomial $c(x)$ modulo x^4+1. In Rijndael, $\mathrm{GF}(2^8)$ is represented by $\mathrm{GF}(2)[t]/m(t)$ where $m(t)$ is $t^8+t^4+t^3+t+1$. The polynomial $c(x)$ is $(t+1)x^3+x^2+x+t$. Thus, we have to be able to multiply elements in $\mathrm{GF}(2^8)$ by only the following fixed polynomials: 1, t, $t+1$. We first create the matrices corresponding to this mutliplication. Note that `i8` (already created) corresponds to multiplying by 1.

```
> multbyt := Matrix( GF(2), 8, 8, [0: i in [1..64]] );
> for i in [1..7] do
>     multbyt[i][i+1] := 1;
> end for;
> multbyt[8][1] := 1;
> multbyt[7][1] := 1;
> multbyt[5][1] := 1;
> multbyt[4][1] := 1;
> multbyt;
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[1 0 0 0 1 0 0 0]
[1 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[1 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0]
> multbytplus1 := multbyt + i8;
```

Let us first create a 32 by 32 matrix which represents multiplying *one* column of the state by $c(x)$ mod x^4+1. Recall from the Rijndael specifications that this is equivalent to multiplying the an input column by the matrix

$$\begin{pmatrix} t & t+1 & 1 & 1 \\ 1 & t & t+1 & 1 \\ 1 & 1 & t & t+1 \\ t+1 & 1 & 1 & t \end{pmatrix}.$$

```
> mix_col := Matrix(GF(2), 32, 32, [0: i in [1..32^2]]);
```

```
> InsertBlock(~mix_col, multbyt, 1, 1);
> InsertBlock(~mix_col, multbytplus1, 1, 9);
> InsertBlock(~mix_col, i8, 1, 17);
> InsertBlock(~mix_col, i8, 1, 25);
> InsertBlock(~mix_col, i8, 9, 1);
> InsertBlock(~mix_col, multbyt, 9, 9);
> InsertBlock(~mix_col, multbytplus1, 9, 17);
> InsertBlock(~mix_col, i8, 9, 25);
> InsertBlock(~mix_col, i8, 17, 1);
> InsertBlock(~mix_col, i8, 17, 9);
> InsertBlock(~mix_col, multbyt, 17, 17);
> InsertBlock(~mix_col, multbytplus1, 17, 25);
> InsertBlock(~mix_col, multbytplus1, 25, 1);
> InsertBlock(~mix_col, i8, 25, 9);
> InsertBlock(~mix_col, i8, 25, 17);
> InsertBlock(~mix_col, multbyt, 25, 25);
```

By inserting the above 32 by 32 matrix in 4 positions along the diaganol of a 128 by 128 matrix, we obatin the MixColumn matrix which corresponds to applying the finite field polynomial multiplication to all four columns of the state.

```
> big_mix_col := Matrix(GF(2),128,128,[0: i in [1..128^2]]);
> InsertBlock(~big_mix_col, mix_col, 1, 1);
> InsertBlock(~big_mix_col, mix_col, 33, 33);
> InsertBlock(~big_mix_col, mix_col, 65, 65);
> InsertBlock(~big_mix_col, mix_col, 97, 97);
>
> linear_diffusion_layer := big_mix_col*linear_diffusion_layer;
```

We can now check that `linear_diffusion_layer` is the same matrix that Murphy and Robshaw obtained. Due to space limitations, we do not print it out again here. The computations of the minimal and characteristic polynomial are done below. It is clear that the exponent of the linear diffusion layer matrix is 16 because the minimal polynomial $m(x)$ times $x + 1$ is equal to $x^{16} + 1$.

```
> time m<x> := MinimalPolynomial(linear_diffusion_layer);
Time: 0.010
> m;
x^15 + x^14 + x^13 + x^12 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 +
    x^3 + x^2 + x + 1
> time c<x> := CharacteristicPolynomial(linear_diffusion_layer);
Time: 0.010
> c;
x^128 + 1
```