Handbook of Magma Functions

Volume 3

Global Arithmetic Fields

John Cannon Wieb Bosma

Claus Fieker Allan Steel

Editors

Version 2.19

Sydney

December 17, 2012

Handbook of Magma Functions

Editors:

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

Handbook Contributors:

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

Production Editors:

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

HTML Production:

Claus Fieker *Allan Steel*

VOLUME 3: OVERVIEW

VOLUME 3: CONTENTS

PART V
LATTICES AND QUADRATIC FORMS

30 LATTICES

Chapter 30
LATTICES

30.1 Introduction

Lattices play an important role in various areas, e.g., representation theory, coding theory, geometry and algebraic number theory.

A lattice in Magma is a free **Z**-module contained in $\mathbf{Q}^n$ or $\mathbf{R}^n$, together with a positive definite inner product having image in **Q** or **R**. The information specifying a lattice is a basis, given by a sequence of elements in $\mathbf{Q}^n$ or $\mathbf{R}^n$, and a bilinear product $(\cdot, \cdot)$, given by $(v, w) = vMw^{tr}$ for a positive definite matrix M.

Central to the lattice machinery in Magma is a fast implementation of the *Lenstra-Lenstra-Lovász lattice reduction* algorithm [LLL82]. The Lenstra-Lenstra-Lovász algorithm (LLL for short) 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 algorithm is based on both the floating-point LLL algorithm of Nguyen and Stehlé [NS09] and the exact integral algorithm as described by de Weger in [dW87], but includes many optimisations, with particular attention to different kinds of input matrices. The algorithm can operate on either a basis matrix or a Gram matrix and can be controlled by many parameters (selection of methods, LLL reduction constants, step and time limits, etc.).

For each lattice, a LLL-reduced basis for the lattice is computed and stored internally when necessary and subsequently used for many operations. This gives maximum efficiency for the operations, yet all the operations are presented using the external ("user") basis of the lattice. Lattices are thus a more efficient alternative to R-spaces where R is the integer ring **Z** since lattices use a LLL-reduced form of the basis matrix extensively instead of the Hermite form of a basis matrix used by R-spaces which may have very large entries.

Another important component of the lattice machinery is a very efficient algorithm for enumerating all vectors of a lattice with norms in a given range. This algorithm is used for computing the minimum, the shortest vectors, vectors up to a chosen length, and vectors close to or closest to a given vector (possibly) outside a lattice.

Several interesting lattices are directly accessible inside Magma using standard constructions (e.g., root lattices and preimages of linear codes). Additionally, an interface has been provided to convert lattices in the database of G. Nebe and N.J.A. Sloane [NS01a] into Magma format.

30.2 Presentation of Lattices

A lattice in Magma is a free **Z**-module of rank m inside $\mathbf{Z}^n$, $\mathbf{Q}^n$, or $\mathbf{R}^n$, where $m \leq n$. We call m the rank or dimension of the lattice and n its degree. A lattice with $m = n$ is called a *full* lattice. There are two pieces of information which completely determine a lattice L: the basis matrix B and the inner product matrix M. M is always a positive definite matrix (i.e., a matrix M such that $vMv^{tr} > 0$ for all non-zero vectors $v \in \mathbf{R}^n$).

All other information associated with a lattice L is derived from B and M. For example, the inner product $(\cdot, \cdot)$ for L is given by $(v, w) = vMw^{tr}$ and thus the Gram matrix F for L giving the inner products of the basis vectors is BMB^{tr}.

Lattices differ from R-spaces in that a lattice is always a **Z**-module even if entries of some elements of the lattice are not integers. For R-spaces the base ring R affects the module structure but for lattices the "base ring" R is just defined to be the smallest ring over which the basis and inner product matrices can be represented. Magma basically has two kinds of lattices: *exact* lattices for which all of the entries of B and M (and thus F) lie in either **Z** or **Q**, and *non-exact* lattices for which all of the entries of B and M (and thus F) lie in a particular approximate real field **R**.

For exact lattices, the "base ring" R may be **Z** or **Q** as mentioned above so that elements are represented over R, but otherwise there is nothing in the lattice which distinguishes between the **Z** and **Q** cases. When exact lattices are printed, the basis matrix B is presented uniquely as an integral matrix divided by a minimal positive integral denominator. The inner product matrix M is presented similarly. This tends to be a more natural presentation which reflects the **Z**-module structure. Another useful feature of this presentation is that the inner product can be efficiently defined to be the identity matrix divided by an appropriate integral scale factor. In this way, the basis can be kept rational while still obtaining an integral primitive Gram matrix (a matrix with integral entries which are coprime), thus avoiding the need for rescaling the basis by an irrational square root. This method is followed in the construction of special lattices provided within Magma.

For non-exact lattices, the basis matrix B is simply presented as it is represented over the ring R which is an approximate real field. Some operations applicable for exact lattices are not possible for non-exact lattices. Thus if in the description of a function below it is said that the function can only be applied to an exact lattice it means that the function cannot be applied to a lattice over an approximate real field.

Two lattices are said to be *compatible* if their inner product matrices are equal. Many operations on lattices assume that their arguments are compatible. Intuitively, two lattices and their elements are not comparable if their inner product matrices are different.

Associated with a lattice L is its coordinate lattice L' which has the standard basis (i.e., its basis matrix is the identity matrix) but has the same Gram matrix as L. Some operations, e.g., automorphism group, always work with the coordinates of elements of the lattice, so it is necessary to move to the coordinate lattice.

The category of lattices is `Lat`.

30.3 Creation of Lattices

Lattices can be created in elementary ways whereby the basis matrix or a generating matrix and possibly also the inner product matrix are specified. MAGMA also provides functions for creating lattices from linear codes or algebraic number fields and for creating some special lattices.

30.3.1 Elementary Creation of Lattices

This subsection describes the elementary ways of creating lattices by supplying the basis or the Gram matrix.

There are two ways of specifying the basis of a lattice at creation. First, a generating matrix can be specified whose rows need not be independent; the matrix is reduced to a LLL-reduced form and zero rows are removed to yield a basis matrix. Secondly, a basis matrix (with independent rows) can be specified; this matrix is used for the basis matrix without any changes being made to it. As well as the basis, a particular inner product can also be specified by a symmetric positive definite matrix. By default (when a particular inner product matrix is not given), the inner product is taken to be the standard Euclidean product.

Instead of giving the basis one can also define a lattice by its Gram matrix F which prescribes the inner products of the basis vectors. The basis is taken to be the standard basis and the inner product matrix is taken as F so that the Gram matrix for the lattice is also F.

`Lattice(X, M)`

`Lattice(n, Q, M)`

`Lattice(S, M)`

`CheckPositive` BOOLELT *Default :* `true`

Construct the lattice L specified by the given generating matrix and with inner product given by the $n \times n$ matrix M. The generating matrix can be specified by an $r \times n$ matrix X, an integer n with a sequence Q of ring elements of length rn (interpreted as a $r \times n$ matrix), or an R-space S of dimension r and degree n. In each case, the generating matrix need not have independent rows (though it always will in the R-space case). The matrix is reduced to a LLL-reduced form and zero rows are removed to yield a basis matrix B of rank m (so $m \leq r$). The lattice L is then constructed to have rank m, degree n, basis matrix B and inner product matrix M. By default MAGMA checks that M is positive definite but by setting `CheckPositive := false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

`Lattice(X)`

`Lattice(n, Q)`

`Lattice(S)`

Construct the lattice L specified by the given generating matrix and with standard Euclidean inner product. The generating matrix can be specified by an $r \times n$ matrix X, an integer n with a sequence Q of ring elements of length rn (interpreted as a $r \times n$ matrix), or an R-space S of dimension r and degree n. In each case, the generating matrix need not have independent rows (though it always will in the R-space case). The matrix is reduced to a LLL-reduced form and zero rows are removed to yield a basis matrix B of rank m (so $m \leq r$). The lattice L is then constructed to have rank m, degree n, basis matrix B and standard Euclidean inner product.

```
LatticeWithBasis(B, M)
LatticeWithBasis(n, Q, M)
LatticeWithBasis(S, M)
```

`CheckIndependent`	BoolElt	*Default :* `true`
`CheckPositive`	BoolElt	*Default :* `true`

Construct the lattice L with the specified $m \times n$ basis matrix and with inner product given by the $n \times n$ matrix M. The basis matrix B can be specified by an $m \times n$ matrix B, an integer n with a sequence Q of ring elements of length mn (interpreted as an $m \times n$ matrix B), or an R-space S of dimension m and degree n (whose basis matrix is taken to be B). The rows of B must be independent (i.e., form a basis). The lattice L is then constructed to have rank m, degree n, basis matrix B and inner product matrix M. (Note that the basis matrix B is *not* reduced to a LLL-reduced form as in the `Lattice` function.) By default Magma checks that the rows of the matrix B specifying the basis are independent but by setting `CheckIndependent := false` this check will be suppressed. By default Magma also checks that M is positive definite but by setting `CheckPositive := false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

```
LatticeWithBasis(B)
LatticeWithBasis(n, Q)
LatticeWithBasis(S)
```

`CheckIndependent`	BoolElt	*Default :* `true`

Construct the lattice L with the specified $m \times n$ basis matrix and with standard Euclidean inner product. The basis matrix B can be specified by an $m \times n$ matrix B, an integer n with a sequence Q of ring elements of length mn (interpreted as an $m \times n$ matrix B), or an R-space S of dimension m and degree n (whose basis matrix is taken to be B). The rows of B must be independent (i.e., form a basis). The lattice L is then constructed to have rank m, degree n, basis matrix B and standard Euclidean product. (Note that the basis matrix B is *not* reduced to a LLL-reduced form as in the `Lattice` function.) By default Magma checks that the rows of the matrix B specifying the basis are independent but by setting `CheckIndependent`

`:= false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

`LatticeWithGram(F)`

`LatticeWithGram(n, Q)`

`CheckPositive` BoolElt *Default :* `true`

Construct a lattice with the given Gram matrix. The Gram matrix F can be specified as a symmetric $n \times n$ matrix F, or an integer n and a sequence Q of ring elements of length n^2 or $\binom{n+1}{2}$. In the latter case, a sequence of length n^2 is regarded as an $n \times n$ matrix and a sequence of length $\binom{n+1}{2}$ as a lower triangular matrix determining a symmetric matrix F.

This function constructs the lattice L of degree n having the standard basis and inner product matrix F, therefore having Gram matrix F as well. By default Magma checks that F is positive definite but by setting `CheckPositive := false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

`StandardLattice(n)`

Create the standard lattice $\mathbf{Z}^n$ with standard Euclidean inner product.

`CoordinateLattice(L)`

Constructs the lattice with the same Gram matrix as the lattice L, but with basis equal to the canonical basis of the free module of Rank(L) over the base ring of L. The identification of basis elements thus determines an isometry of lattices.

`ScaledLattice(L,n)`

Constructs the coordinate lattice with Gram matrix equal to that of the lattice L scaled by the integer or rational n. The identification of basis elements thus determines an similitude of lattices.

Example H30E1

We create the lattice in $\mathbf{Z}^3$ of degree 3 and rank 2 generated by the vectors $(1, 2, 3)$ and $(3, 2, 1)$ in four different ways. The first two (identical) lattices are represented by LLL-reduced bases since the `Lattice` function is used to create them, while the latter two (identical) lattices remain represented by the original basis since the `LatticeWithBasis` function is used to create them.

```
> B := RMatrixSpace(IntegerRing(), 2, 3) ! [1,2,3, 3,2,1];
> B;
[1 2 3]
[3 2 1]
> L1 := Lattice(B);
> L1;
Lattice of rank 2 and degree 3
Basis:
( 2  0 -2)
( 1  2  3)
> L2 := Lattice(3, [1,2,3, 3,2,1]);
> L2 eq L1;
true
> L3 := LatticeWithBasis(B);
> L3;
Lattice of rank 2 and degree 3
Basis:
(1 2 3)
(3 2 1)
> L4 := LatticeWithBasis(3, [1,2,3, 3,2,1]);
> L4 eq L3, L1 eq L3;
true true
```

We next create a 3×3 positive definite matrix M and create the lattice L with basis the same as above but also with inner product matrix M. We note the Gram matrix of the lattice which equals BMB^{tr} where B is the basis matrix.

```
> B := RMatrixSpace(IntegerRing(), 2, 3) ! [1,2,3, 3,2,1];
> M := MatrixRing(IntegerRing(), 3) ! [3,-1,1, -1,3,1, 1,1,3];
> M;
[ 3 -1  1]
[-1  3  1]
[ 1  1  3]
> IsPositiveDefinite(M);
true
> L := LatticeWithBasis(B, M);
> L;
Lattice of rank 2 and degree 3
Basis:
(1 2 3)
(3 2 1)
Inner Product Matrix:
[ 3 -1  1]
[-1  3  1]
[ 1  1  3]
> GramMatrix(L);
[56 40]
[40 40]
```

Finally, we create a lattice C with the same Gram matrix as the last lattice, but with standard basis. To do this we use the `LatticeWithGram` function. This lattice C is the *coordinate* lattice of L: it has the same Gram matrix as L but is not compatible with L since its inner product matrix is different to that of L.

```
> F := MatrixRing(IntegerRing(), 2) ! [56,40, 40,40];
> C := LatticeWithGram(F);
> C;
Standard Lattice of rank 2 and degree 2
Inner Product Matrix:
```

```
[56 40]
[40 40]
> GramMatrix(C);
[56 40]
[40 40]
> C eq CoordinateLattice(L);
true
> GramMatrix(C) eq GramMatrix(L);
true
>  C eq L;
Runtime error in 'eq': Arguments are not compatible
```

30.3.2 Lattices from Linear Codes

This subsection presents some standard constructions (known as constructions ‘A’ and ‘B’) to obtain lattices from linear codes. In some cases the structural invariants of these lattices (e.g., minimum and kissing number, hence the centre density) can be deduced from invariants of the codes; in general one still gets estimates for the invariants of the lattices.

`Lattice(C, "A")`

Let C be a linear code of length n over a prime field $K := \mathbf{F}_p$. Construct the lattice $L \subseteq \mathbf{Z}^n$ which is the full preimage of C under the natural epimorphism from $\mathbf{Z}^n$ onto K^n.

`Lattice(C, "B")`

Let C be a linear code of length n over a prime field $K := \mathbf{F}_p$ such that all code words have coordinate sum 0. Construct the lattice $L \subseteq \mathbf{Z}^n$ which consists of all vectors reducing modulo p to a code word in C and whose coordinate sum is 0 modulo p^2. The inner product matrix is set to the appropriate scalar matrix so that the Gram matrix is integral and primitive (its entries are coprime).

Example H30E2

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
```

30.3.3 Lattices from Algebraic Number Fields

Let K be an algebraic number field of degree n over $\mathbf{Q}$. Then K has n embeddings into the complex numbers, denoted by $\sigma_1, \ldots, \sigma_n$. By convention the first s of these are embeddings into the real numbers and the last $2t$ are pairs of complex embeddings such that $\sigma_{s+t+i} = \overline{\sigma_{s+i}}$ for $1 \le i \le t$. The T_2-norm $K \to \mathbf{R} : \alpha \mapsto \sum_{i=1}^{n} |\sigma_i(\alpha)|^2$ defines a positive definite norm on K in which an order or ideal is a positive definite lattice. We restrict to the r real embeddings and the first s complex embeddings to obtain an embedding of a rank n order or ideal $\mathbf{R}^s \times \mathbf{C}^s = \mathbf{R}^n$. By rescaling the complex coordinates by $\sqrt{2}$, we recover the T_2-norm of the element as its Euclidean norm under the embedding in $\mathbf{R}^n$. This embedding, given by

$$\alpha \mapsto \Big(\sigma_1(\alpha), \ldots, \sigma_s(\alpha), \sqrt{2}\,\mathrm{Re}(\sigma_{s+1}(\alpha)), \sqrt{2}\,\mathrm{Im}(\sigma_{s+1}(\alpha)), \ldots, \sqrt{2}\,\mathrm{Im}(\sigma_{s+t}(\alpha))\Big),$$

is called the *Minkowski map* on K.

`MinkowskiLattice(O)`

`Lattice(O)`

Given an order O in a number field of degree n, create the lattice L in $\mathbf{R}^n$ generated by the images of the basis of O under the Minkowski map, followed by the isomorphism $O \to L$.

`MinkowskiLattice(I)`

`Lattice(I)`

Given an ideal I of an order O in a number field of degree n, create the lattice in $\mathbf{R}^n$ generated by the images of the basis of I under the Minkowski map, followed by the isomorphism $O \to L$.

`MinkowskiSpace(K)`

Construct the real inner product space V defined with respect to the inner product matrix of the T_2-norm on the number field K, followed by the Minkowski map $K \to V$.

Example H30E3

In this example we create the lattice corresponding to the equation order of the number field $\mathbf{Q}(\sqrt[3]{15})$. For comparison we apply the Minkowski map to the elements of the basis for the order.

```
> P<x> := PolynomialRing(Integers());
> R := EquationOrder(NumberField(x^3-15));
> w := R![0,1,0];
> L, f := MinkowskiLattice(R);
> L;
Lattice of rank 3 and degree 3 over Real Field
Basis:
Lattice of rank 3 and degree 3 over Real field of precision 30
Basis:
(1.00000000000000000000000000000 1.41421356237309504880168872421
    0.000000000000000000000000000000)
(2.46621207433047010149161132315 -1.74387528160321720079629694216
    3.02048058980025565688335869988)
(6.08220199557340018489844499774 -4.30076627561630297914843739812
    -7.44914570084621027635727295139)
> B := Basis(R);
> f(B[2]);
> f(B[2]);
(2.46621207433047010149161132315 -1.74387528160321720079629694216
    3.02048058980025565688335869988)
```

Similarly, we create the lattice defined by the ideal generated by $w^2 + 1$, and verify the that the T_2-norm, given by `Length` on the number field, agrees with the norm of the lattice image.

```
> I := ideal< R | w^2+1 >;
> L, f := Lattice(I);
> B := Basis(I);
> [ Length(B[k]) : k in [1..3] ];
[ 113.9795433448711545671225095419565069230968018982624192080454
3346320196288153038584473344161989557225281060164140817511930289
049332599978630989750057117093892447723724,
693.246605986720200554695334993213407259633012131529936960318255
6345313709169384387254042929336529027528589276741384648056766427
8812020993502245102110970371559009076524,
4216.46589035691627937357288301497314034052453149249823527965295
1232760419192679099060699351691798691646069328322562756396547517
8203732351663613772502545069466948994156 ]
> [ Norm(f(B[k])) : k in [1..3] ];
[ 113.979543344871154567122509542,
693.246605986720200554695334993, 4216.46589035691627937357288302
]
```

30.3.4 Special Lattices

This subsection presents functions to construct some special lattices, namely root lattices, laminated lattices and Kappa-lattices.

A much wider variety of lattices can be found in a database provided by G. Nebe and N.J.A. Sloane at the web-site [NS01a]. These lattices can easily be made accessible to Magma by a conversion script also available at this web-site. The database currently contains (at least):

- The root lattices A_n and their duals for $1 \leq n \leq 24$
- The root lattices D_n and their duals for $1 \leq n \leq 24$
- The root lattices E_n and their duals for $6 \leq n \leq 8$
- The laminated lattices Λ_n for $1 \leq n \leq 24$ including the 16-dimensional Barnes-Wall lattice Λ_{16} and the Leech lattice Λ_{24}
- The Kappa-lattices K_n and their duals for $7 \leq n \leq 13$ including the Coxeter-Todd lattice K_{12}
- The perfect lattices up to dimension 7
- The 3-dimensional Bravais lattices
- Various interesting lattices in dimensions 20, 24, 28, 32, 40, 80, 105 including, e.g., some of the densest known lattices in dimension 32.

`Lattice(X, n)`

Given a family name X as a string which is one of `"A"`, `"B"`, `"C"`, `"D"`, `"E"`, `"F"`, `"G"`, `"Kappa"` or `"Lambda"`, together with an integer n, construct a lattice subject to the following specifications:

`A`: The root lattice A_n which is the zero-sum lattice in $\mathbf{Q}^{n+1}$.

`B`: $n \geq 2$: The root lattice B_n which is the standard lattice of dimension n.

`C`: $n \geq 3$: The root lattice C_n which is the even sublattice of $\mathbf{Z}^n$ and is equal to D_n.

`D`: $n \geq 3$: The root lattice D_n which is the even sublattice of $\mathbf{Z}^n$, also called the checkerboard lattice.

`E`: $6 \leq n \leq 8$: The root lattice E_n, also called Gosset lattice.

`F`: $n = 4$: The root lattice F_4 which is equal to D_4.

`G`: $n = 2$: The root lattice G_2 which is equal to A_2.

`Kappa`: $1 \leq n \leq 13$: The Kappa-lattice K_n. For $n = 12$ this is the Coxeter-Todd lattice.

`Lambda`: $1 \leq n \leq 31$: The laminated lattice Λ_n. For $n = 16$ this is the Barnes-Wall lattice, for $n = 24$ the Leech lattice.

To avoid irrational entries, each lattice is presented with inner product matrix taken to be the identity matrix divided by a suitable scale factor so that the Gram matrix is integral and primitive (its entries are coprime).

30.4 Lattice Elements

The following functions allow basic operations on elements of lattices. The elements of lattices are simply (row) vectors, just as for R-spaces. Most of the operations for R-space elements are also applicable to lattice elements.

30.4.1 Creation of Lattice Elements

`L . i`

Return the i-th basis element of the current basis of the lattice L.

`L ! Q`

`elt< L | Q >`

Given a lattice L of degree n and a sequence Q of length n, create the lattice element with the corresponding sequence elements as entries. The sequence must consist of elements coercible into the base ring of L. The resulting vector must lie within L.

`CoordinatesToElement(L, C)`

`Coordelt(L, C)`

Given a lattice L of rank m and a sequence or vector $C = [c_1, \ldots, c_m]$ of length m of *integers*, create the lattice element $c_1 \cdot b_1 + \ldots + c_m \cdot b_m$, where $[b_1, \ldots, b_m]$ is the basis of L.

`L ! 0`

`Zero(L)`

Return the zero element of the lattice L.

30.4.2 Operations on Lattice Elements

`-v`

Given an element v in a lattice L, return its negation $-v$ in L.

`v + w`

Given elements v and w in a lattice L, return the sum $v + w$ in L.

`v - w`

Given elements v and w in a lattice L, return the difference $v - w$ in L.

`v * s`

`s * v`

Given an element v in a lattice L and a scalar s of the ring S, return the product $s \cdot v$ (scalar multiplication of v by s). If s is an integer, the resulting vector will lie in L; otherwise the resulting vector will lie in the R-space of the appropriate degree whose coefficient ring is the parent S of s.

`v / s`

Given an element v in a lattice L and a scalar s of the ring S, return the product $(1/s) \cdot v$ (scalar multiplication of v by $1/s$). The resulting vector will always lie in the R-space of the appropriate degree whose coefficient ring is the field of fractions of the parent S of s.

`v div d`

Given an element v in a lattice L and an *integer* d, return the vector $(1/d) \cdot v$ (scalar multiplication of v by $1/d$) as an element of L if the scaled vector lies in L. If the scaled vector does not lie in L, an error ensues. Note that this is different from v/d.

`v +:= w`

(Assignment statement.) Replace lattice element v by the sum $v + w$.

`v -:= w`

(Assignment statement.) Replace lattice element v by the difference $v - w$.

`v *:= n`

(Assignment statement.) Replace lattice element v by the scalar product $n \cdot v$, where n is an integer.

`v * T`

Given an element v in a lattice L of degree n, return the result of multiplying v from the right by the $n \times n$ matrix T. The matrix T may be any matrix which is n by n and over the base ring of L. The resulting product must lie in the lattice L.

`InnerProduct(v, w)`

`(v, w)`

Given elements v and w of a lattice L, return their inner product (v, w) with respect to the inner product of L. This is vMw^{tr} where M is the inner product matrix of L.

`Norm(v)`

Given an element v of a lattice L, return its norm (v, v) with respect to the inner product of L. This is vMv^{tr} where M is the inner product matrix of L. Note that in the case of a lattice with standard Euclidean inner product this is the square of the usual Euclidean length.

```
Length(v, K)
```

```
Length(v)
```

Given an element v of a lattice L, return its length $\sqrt{(v,v)}$ with respect to the inner product of L as an element of the real field K. This is $\sqrt{vMv^{tr}}$ where M is the inner product matrix of L. The argument for the real field K may be omitted, in which case K is taken to be the current default real field. In the case of a lattice with standard Euclidean inner product this is the usual Euclidean length.

```
Support(v)
```

Given an element v of a lattice L, return its support, i.e., the numbers of the columns at which v has non-zero entries.

30.4.3 Predicates and Boolean Operations

```
v in L
```

Given an element v of a lattice which is compatible with the lattice L, return `true` if and only if v is in L.

```
v eq w
```

Given elements v and w of a lattice L, return `true` if and only if lattice elements v and w of lattice L are equal.

```
v ne w
```

Given elements v and w of a lattice L, return `false` if and only if lattice elements v and w of lattice L are equal.

```
IsZero(v)
```

Given an element v of a lattice L, return `true` if and only if v is the zero element L.

30.4.4 Access Operations

```
ElementToSequence(v)
```

```
Eltseq(v)
```

Given an element v of a lattice L of degree n, return the sequence of entries of v of length n.

```
Coordinates(v)
```

Given an element v of a lattice L having **Z**-basis $[b_1, \ldots, b_m]$, return a sequence $[c_1, \ldots, c_m]$ of elements of **Z** giving the (unique) coordinates of v relative to the **Z**-basis, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

`Coordinates(L, v)`

Given a lattice L of degree n having **Z**-basis $[b_1, \ldots, b_m]$, together with an element v of a lattice L' (also of degree n), return a sequence $[c_1, \ldots, c_m]$ of elements of **Z** giving the (unique) coordinates of v relative to the **Z**-basis of L, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

`CoordinateVector(v)`

Given an element v of a lattice L having **Z**-basis $[b_1, \ldots, b_m]$, return the vector $c = (c_1, \ldots, c_m)$ of the coordinate lattice C of L (see the function `CoordinateLattice`) giving the (unique) coordinates of v relative to the **Z**-basis, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

`CoordinateVector(L, v)`

Given a lattice L of degree n having **Z**-basis $[b_1, \ldots, b_m]$, together with an element v of a lattice L' (also of degree n), return the vector $c = (c_1, \ldots, c_m)$ of the coordinate lattice C of L (see the function `CoordinateLattice`) giving the (unique) coordinates of v relative to the **Z**-basis of L, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

Example H30E4

This example demonstrates simple uses of the operations on lattice elements.

```
> L := LatticeWithBasis(3, [1,0,0, 1,2,3, 3,6,2]);
> L;
Lattice of rank 3 and degree 3
Basis:
(1 0 0)
(1 2 3)
(3 6 2)
> Coordelt(L, [1, 2, 1]);
( 6 10  8)
> v := L.2;
> w := L ! [2, 4, 6];
> Eltseq(v);
[ 1, 2, 3 ]
> Coordinates(w);
[ 0, 2, 0 ]
> Coordelt(L, [1, 1, 1]);
(5 8 5)
> Norm(v);
14
> InnerProduct(v, w);
28
> A := MatrixRing(Integers(), 3);
> X := A ! [0,-1,0, 1,0,0, 0,1,2];
> X;
[ 0 -1  0]
[ 1  0  0]
[ 0  1  2]
> u := L.1 + L.3;
> Determinant(X);
2
> Norm(u);
56
> u * X;
( 6 -2  4)
> Norm(u * X);
56
```

30.5 Properties of Lattices

The following functions provide access to the elementary attributes and properties of lattices. Other attributes and invariants of a lattice, e.g., successive minimum, kissing number, and theta series are described in subsequent sections.

30.5.1 Associated Structures

`AmbientSpace(L)`

The ambient rational or real vector space in which the lattice L embeds, followed by the embedding map.

`CoordinateSpace(L)`

The ambient vector space of the coordinate lattice, i.e., the vector space of dimension equal to the rank of the lattice L, with inner product matrix equal to the Gram matrix of L. The embedding map is returned as the second return value.

`Category(L)`

`Type(L)`

Returns the category `Lat` of lattices.

30.5.2 Attributes of Lattices

`Dimension(L)`

`Rank(L)`

Return the rank of the lattice L, which equals the number of basis elements in L. Note that the rank of the lattice may be smaller than its degree n, which is the dimension of the real space $\mathbf{R}^n$ in which L is defined.

`Degree(L)`

Return the degree of the lattice L, which is the dimension n of the real space $\mathbf{R}^n$ in which L is defined.

`Degree(v)`

Return the degree of the lattice element v, which the degree of the lattice to which it belongs.

`Content(L)`

Given an exact lattice L, return the largest rational number c such that $(u, v) \in c\mathbf{Z}$ for all $u, v \in L$.

`Level(L)`

Given an integral lattice L, return the smallest integer k such that $k(v, v) \in 2\mathbf{Z}$ for all v in the dual of L.

`Determinant(L)`

Returns the determinant of the lattice L, which is defined to be the determinant of the Gram matrix F of L. For a full rank lattice the square root of `Determinant(L)` is the volume of a fundamental parallelotope of the lattice.

`GramMatrix(L)`

Return the Gram matrix for the lattice L of rank m, which is the $m \times m$ matrix $F = BMB^{tr}$, where B is the basis matrix of L and M is the inner product matrix of L. Thus the (i, j)-th entry of F equals the inner product of the basis vectors b_i and b_j of L.

`GramMatrix(X)`

Given a matrix X, return XX^{tr}. Note that this function will take half the time as would be taken for the invocation `X*Transpose(X)` since the symmetry of the result is taken advantage of.

`InnerProductMatrix(L)`

The inner product matrix M of the lattice L, which is an $n \times n$ matrix, where n is the degree of L. If L has the standard Euclidean product, M is the identity matrix.

```
Basis(L)
```

Return the basis of the lattice L as a sequence $[b_1, \dots, b_m]$ of elements of L.

```
BasisMatrix(L)
```

Return the $m \times n$ matrix having the basis elements of L as rows, where m is the rank and n the degree of L. The coefficient ring of the matrix is the same as the base ring of L.

```
BasisDenominator(L)
```

Given an exact lattice L, return the common denominator of the entries of the current basis of L.

```
QuadraticForm(L)
```

The quadratic form of the lattice L as a multivariate polynomial.

30.5.3 Predicates and Booleans on Lattices

```
L eq M
```

Given lattices L and M, return `true` if and only if the lattices have the same basis matrix and inner product matrix.

```
L ne M
```

The logical negation of `eq`.

```
L subset M
```

Return `true` if and only if L is a sublattice of M, i.e., both L and M are lattices, and L is a subset of M.

```
IsExact(L)
```

Return `true` if and only if L is an exact lattice, i.e., the coefficient ring of L is $\mathbf{Z}$ or $\mathbf{Q}$ and $(v, w) \in \mathbf{Q}$ for all $v, w \in L$.

```
IsIntegral(L)
```

Return `true` if and only if L is an integral lattice, i.e., if and only if $(v, w) \in \mathbf{Z}$ for all $v, w \in L$.

```
IsEven(L)
```

Return `true` if and only if L is an even lattice, i.e., if and only if L is integral and $(v, v) \in 2\mathbf{Z}$ for all $v \in L$.

30.5.4 Base Ring and Base Change

```
BaseRing(L)
```
```
CoefficientRing(L)
```

Return the base ring of the lattice L, which is the ring over which the elements of L are represented. Note that lattices are always **Z**-modules even if the base ring is not **Z**; the base ring R is just defined to be the smallest ring over which the basis and inner product matrices can be represented. See the section on presentation of lattices at the beginning of the chapter for further discussion.

```
CoordinateRing(L)
```

Return the ring of coordinate coefficients for the lattice L. This currently will always return the integer ring **Z**.

```
ChangeRing(L, S)
```
```
BaseChange(L, S)
```
```
BaseExtend(L, S)
```

Given a lattice L, return the lattice L' obtained from coercing the entries of the basis and of the matrix for the inner product into the ring S, together with the homomorphism from L to L'. This will result in an error if any of the entries is not coercible into S. This function is only really useful for moving between real fields of varying precision (it is unnecessary and ineffectual to change a lattice from **Z** to **Q**; see the section on presentation of lattices).

30.6 Construction of New Lattices

30.6.1 Sub- and Superlattices and Quotients

```
sub< L | S >
```

Given a lattice L and a list S, construct the sublattice L' of L generated by the elements specified by the list S. Each term S_i of the list S must be an expression defining an object of one of the following types:

(a) an element of L or coercible into L,

(b) a set or sequence of elements coercible into L,

(c) a sublattice of L,

(d) a set or sequence of sublattices of L.

The constructor returns the sublattice L' and the inclusion homomorphism from L' into L.

```
ext< L | S >
```

Given a lattice L lying inside $V = \mathbf{R}^n$ and a list S, construct the superlattice L' of L generated by L together with the elements specified by the list S. Each term S_i of the list S must be an expression defining an object of one of the following types:

(a) an element of V or coercible into V,

(b) a set or sequence of elements coercible into V,

(c) a sublattice of V,

(d) a set or sequence of sublattices of V.

The constructor returns the superlattice L' and the inclusion homomorphism from L into L'.

```
T * L
```

Given a lattice L of rank m and an $l \times m$ integer matrix, construct the sublattice of L defined by the transformation matrix T, i.e., the lattice generated by the rows of the matrix obtained by multiplying the basis matrix of L from the left by T. The resulting lattice will have rank less than or equal to l.

```
s * L
L * s
```

Given a lattice L and a scalar s, construct the sublattice or superlattice of L obtained by multiplying the basis matrix of L by the scalar s.

```
L / s
```

Given a lattice L and a scalar s, construct the sublattice or superlattice of L obtained by multiplying the basis matrix of L by the scalar $1/s$.

```
quo< L | S >
```

Given a lattice L and a list S, construct the quotient L/L', where L' is the sublattice of L generated by the elements of the list S. The elements of S must be the same as for the `sub<>` constructor. The quotient $Q := L/L'$ is constructed as an abelian group. As a second value the function returns the natural epimorphism $L \to Q$.

```
L / S
```

Given two lattices L and S such that S is a sublattice of L, construct the quotient $Q := L/S$ as an abelian group. As a second value the function returns the natural epimorphism $L \to Q$.

```
Index(L, S)
```

Given a lattice L and a sublattice S of L, return the index of S in L. This is the cardinality of the quotient L/S. If the index is infinite, zero is returned.

Example H30E5

We demonstrate simple uses of the `sub`-, `ext`- and `quo`-constructors.

```
> L := LatticeWithBasis(4, [1,2,3,4, 0,1,1,1, 0,1,3,5]);
> L;
Lattice of rank 3 and degree 4
Basis:
(1 2 3 4)
(0 1 1 1)
(0 1 3 5)
> E := ext<L | [1,0,0,0]>;
> E;
Lattice of rank 3 and degree 4
Basis:
( 1  0  0  0)
( 0  1  0 -1)
( 0  1  1  1)
> Index(E, L);
2
> Q, f := quo<E | L>;
> Q;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*Q.1 = 0
> f(E.1);
Q.1
```

30.6.2 Standard Constructions of New Lattices

The functions in this section enable one to construct new lattices from old ones using standard operations. Note that the functions will preserve the basis matrices of their arguments if possible (e.g., `DirectSum`), but if this is not possible the basis of the resulting lattice will be LLL-reduced (e.g., +, `meet`).

`Dual(L)`

`Rescale` BoolElt *Default :* `true`

Let L be a lattice in $\mathbf{R}^n$ and let $V := \mathbf{R} \otimes_{\mathbf{Z}} L$ be the real vector space generated by L. Then the dual lattice $L^\#$ of L is defined by $L^\# := \{v \in V | (v, l) \in \mathbf{Z}\ \forall l \in L\}$. For an integral lattice L one always has $L \subseteq L^\#$. This function returns a rescaled version L' of the dual $L^\#$ by default so that the basis of L' is LLL-reduced and L' is an integral lattice and its Gram matrix is primitive (its entries are coprime). By setting the parameter `Rescale` to `false`, the rescaling can be suppressed and the proper dual lattice is returned.

`PartialDual(L, n)`

Rescale BoolElt *Default :* `true`

Given an integral lattice L and a positive integer n that divides the exponent e of the `DualQuotient` group of L, this function computes the nth partial dual of L. This is defined by pulling back $(e/n) \cdot g$ for generators g of the `DualQuotient` and intersecting with the lattice itself.

`DualBasisLattice(L)`

Let L be a lattice in $\mathbf{R}^n$ and let $V := \mathbf{R} \otimes_{\mathbf{Z}} L$ be the real vector space generated by L, then $L^\# := \{v \in V | (v, l) \in \mathbf{Z} \ \forall l \in L\}$ is the dual lattice of L. Let B be the basis matrix of L, M the inner product matrix of L, and F the Gram matrix of L. This function returns the dual $L^\#$ as the lattice with basis matrix $F^{-1}B$ and inner product matrix M (so that its Gram matrix is F^{-1}).

`DualQuotient(L)`

Given an integral lattice L, construct the dual quotient Q of L, which is defined to be the finite abelian group $L^\#/L$ of order `Determinant(L)`, where $L^\#$ is the unscaled dual lattice of L (the lattice returned by `Dual` with the `Rescale` parameter set to `false`). This function returns three values:

(a) The dual quotient Q.

(b) The unscaled dual lattice $L^\#$.

(c) The natural epimorphism $\phi : L^\# \to Q$ whose kernel is L.

`EvenSublattice(L)`

Given an integral lattice L, construct its maximal even sublattice together with the natural embedding into L.

Example H30E6

```
> L := Lattice(LatticeDatabase(), 541);
> Dimension(L);
29
> IsEven(L);
true
> IsEven(Dual(L));
false;
> G := DualQuotient(L);
> Exponent(G);
8
> Factorization(Determinant(L));
[ <2, 31> ]
> PartialDual(L,1) eq L;
true
> Factorization(Determinant(PartialDual(L, 2)));
```

```
[ <2, 54> ]
> Factorization(Determinant(PartialDual(L, 4)));
[ <2, 73> ]
> Factorization(Determinant(PartialDual(L, 8)));
[ <2, 56> ]
> Factorization(Determinant(Dual(L)));
[ <2, 56> ]
```

`L + M`

Given compatible lattices L and M, construct the lattice generated by their union.

`L meet M`

Given compatible lattices L and M, construct their intersection $L \cap M$.

`DirectSum(L, M)`

`OrthogonalSum(L, M)`

Given lattices L and M, construct their orthogonal sum which is their direct sum with inner product being the orthogonal sum of the inner products of L and M.

`OrthogonalDecomposition(L)`

Given a lattice L, construct the sequence of indecomposable orthogonal summands composing L. Additional bilinear forms can be given in F. In this case the decomposition will be orthogonal wrt. these forms as well.

`OrthogonalDecomposition(F)`

`Optimize` BoolElt *Default :* `false`

Given a sequence of bilinear forms F, where the first form is positive definite, returns the basis matrices $B_1, \ldots, B_s$ of the indecomposable orthogonal summands of the standard lattice $\mathbf{Z}^n$ wrt. the forms in F. The second return value is a list of s sequences. The i-th sequence contains the forms of F wrt. to basis described by B_i. If `Optimize` is set, then the basis matrices will be LLL reduced wrt. the first form in F.

`TensorProduct(L, M)`

Given two lattices L and M, construct their tensor product with inner product given by the Kronecker product of the matrices defining the inner products of L and M.

`ExteriorSquare(L)`

Given a lattice L, construct its exterior square, generated by the skew tensors in $L \otimes L$. The inner product is inherited from the inner product of the tensor square of the vector space containing L and the exterior square lattice lies inside the tensor square of the lattice.

`SymmetricSquare(L)`

Given a lattice L, construct its symmetric square, generated by the symmetric tensors in $L \otimes L$. The inner product is inherited from the inner product of the tensor square of the vector space containing L and the symmetric square lattice lies inside the tensor square of the lattice.

`PureLattice(L)`

Given a lattice L of degree n with integral or rational entries, return the pure lattice $P = (\mathbf{Q} \otimes L) \cap \mathbf{Z}^n$ of L. The pure lattice P generates the same subspace in $\mathbf{Q}^n$ over $\mathbf{Q}$ that L does but the elementary divisors of its basis matrix are trivial.

`IntegralBasisLattice(L)`

Given an exact lattice L, return the lattice obtained from L by multiplying the basis by the smallest positive scalar S so that the resulting basis is integral, and S.

30.7 Reduction of Matrices and Lattices

The functions in this section perform reduction of lattice bases. For each reduction algorithm there are three functions: a function which takes a basis matrix, a function which takes a Gram matrix and a function which takes a lattice.

30.7.1 LLL Reduction

The Lenstra-Lenstra-Lovász algorithm [LLL82] was first described in 1982 and was immediately used by the authors to provide a polynomial-time algorithm for factoring integer polynomials, for solving simultaneous diophantine equations and for solving the integer programming problem. It very quickly received much attention and in the last 25 years has found an incredible range of applications, in such areas as computer algebra, cryptography, optimisation, algorithmic number theory, group theory, etc. However, the original LLL algorithm is mainly of theoretical interest, since, although it has polynomial time complexity, it remains quite slow in practice. Rather, floating-point variants are used, where the underlying Gram-Schmidt orthogonalisation is performed with floating-point arithmetic instead of rational arithmetic (which produces huge numerators and denominators). Most of these floating-point variants are heuristic, but here we use the provable Nguyen-Stehlé algorithm [NS09] (see also [Ste09] for more details about the implementation).

Let $\delta \in (1/4, 1]$ and $\eta \in [1/2, \sqrt{\delta})$. An ordered set of d linearly independent vectors $b_1, b_2, \ldots, b_d \in \mathbf{R}^n$ is said to be (δ, η)-LLL-reduced if the two following conditions are satisfied:

(a) For any $i > j$, we have $|\mu_{i,j}| \leq \eta$,

(b) For any $i < d$, we have $\delta \|b_i^*\|^2 \leq \|b_{i+1}^* + \mu_{i+1,i} b_{i+1}^*\|^2$,

where $\mu_{i,j} = (b_i, b_j^*)/\|b_j^*\|^2$ and b_i^* is the i-th vector of the Gram-Schmidt orthogonalisation of $(b_1, b_2, \ldots, b_d)$. LLL-reduction classically refers to the case where $\eta = 1/2$ and $\delta = 3/4$ since it was the choice of parameters originally made in [LLL82], but the closer that η

and δ are to $1/2$ and 1, respectively, the more reduced the lattice basis should be. In the classical LLL algorithm, the polynomial-time complexity is guaranteed as long as $\delta < 1$, whereas the floating-point LLL additionally requires that $\eta > 1/2$.

A (δ, η)-LLL-reduced basis $(b_1, b_2, \ldots, b_d)$ of a lattice L has the following useful properties:

(a) $\|b_1\| \leq \left(\delta - \eta^2\right)^{-(d-1)/4} (\det L)^{1/d}$,

(b) $\|b_1\| \leq (\delta - \eta^2)^{-(d-1)/2} \min_{b \in L\setminus\{0\}} \|b\|$,

(c) $\prod_{i=1}^d \|b_i\| \leq (\delta - \eta^2)^{-d(d-1)/4} (\det L)$,

(d) For any $j < i$, $\|b_j^*\| \leq (\delta - \eta^2)^{(j-i)/2} \|b_i^*\|$.

The default Magma parameters are $\delta = 0.75$ and $\eta = 0.501$, so that $(\delta - \eta^2)^{-1/4} < 1.190$ and $(\delta - \eta^2)^{-1/2} < 1.416$. The four previous bounds can be reached, but in practice one usually obtains better bases. It is possible to obtain lattice bases satisfying these four conditions without being LLL-reduced, by using the so-called Siegel condition [Akh02]; this useful variant, though available, is not the default one.

Internally, the LLL routine may use up to eight different LLL variants, one of them being de Weger's exact integer method [dW87], and the seven others relying upon diverse kinds of floating-point arithmetic. All but one of these variants are heuristic and implement diverse ideas from [SE94], Victor Shoup's NTL [Sho] and [NS06]. The heuristic variants possibly loop forever, but when that happens it is hopefully detected and a more reliable variant is used. For a given input, the LLL routine tries to find out which variant should be the fastest, and may eventually call other variants before the end of the execution: either because it deems that the current variant loops forever or that a faster variant could be used with the thus-far reduced basis. This machinery remains unknown to the user (except when the LLL verbose mode is activated) and makes the Magma LLL routine the fastest currently available, with the additional advantage of being fairly reliable.

The floating-point variants essentially differ on the two following points:

(a) whether or not the matrix basis computations are performed with the help of a complete knowledge of the Gram matrix of the vectors (the matrix of the pairwise inner products),

(b) the underlying floating-point arithmetic.

In theory, to get a provable variant, one needs the Gram matrix and arbitrary precision floating-point numbers (as provided by the MPFR-based [Pro] Magma real numbers). In practice, to get an efficient variant, it is not desirable not to maintain the exact knowledge of the Gram matrix (maintaining the Gram matrix represents asymptotically a large constant fraction of the total computational cost), while it is desirable to use the machine processor floating-point arithmetic (for instance, C doubles). Three of the seven variants use the Gram matrix, while the four others do not. In each group (either the three or the remaining four), one variant relies on arbitrary precision floating-point arithmetic, another on C doubles and another on what we call doubles with extended exponent. These last variants are important because they are almost as fast as the variants based on C doubles, and can be used for much wider families of inputs: when the initial basis is made of integers larger than approximately 500 bits, overflows can occur with the C doubles. To be precise, a "double with extended exponent" is a C double with a C integer, the last one being the extended exponent. So far, we have described six variants. The seventh does not rely on the Gram matrix and is based on an idea similar to "doubles with extended exponents". The difference is that for a given vector of dimension n, instead of having n pairs of doubles and integers, one has n doubles and only one integer: the extended exponent is "factorised". This variant is quite often the one to use in practice, because it is reasonably efficient and not too unreliable.

Important warning. By default, the `LLL` routine is entirely reliable. This implies the default provable variant can be *much* slower than the heuristic one. By setting `Proof` to `false`, a significant run-time improvement can be gained without taking much risk: even in that case, the `LLL` routine remains more reliable than any other implementation based on floating-point arithmetic. For any application where guaranteed LLL-reduction is not needed, we recommend setting `Proof` to `false`.

Recommended usage. The formal description of the function `LLL` below explains all the parameters in detail, but we first note here some common situations which arise. The desired variant of LLL is often not the default one. Since the LLL algorithm is used in many different contexts and with different output requirements, it seems impossible to define a natural default variant, and so using the LLL routine efficiently often requires some tuning. Here we consider three main-stream situations.

- It may be desired to obtain the main LLL inequalities (see the introduction of this subsection), without paying much attention to the δ and η reduction parameters. In this case one should activate the `Fast` parameter, possibly with the verbose mode to know afterwards which parameters have been used.
- It may be desired to have the main LLL inequalities for a given pair of parameters (δ, η). In this case one should set the parameters `Delta` and `Eta` to the desired values, and set `SwappingCondition` to `"Siegel"`.
- It may be desired to compute a very well LLL-reduced basis. In this case one should set `Delta` to 0.9999, `Eta` to 0.5001, and possibly also activate the `DeepInsertion` option.

In any case, if you want to be absolutely sure of the quality of the output, you need to keep the `Proof` option activated.

Example H30E7

This example is meant to show the differences between the three main recommended usages above. Assume that we are interested in the lattice generated by the rows of the matrix B defined as follows.

```
> R:=RealField(5);
> B:=RMatrixSpace(IntegerRing(), 50, 51) ! 0;
> for i := 1 to 50 do B[i][1] := RandomBits(10000); end for;
> for i := 1 to 50 do B[i][i+1] := 1; end for;
```

The matrix B is made of 100 vectors of length 101. Each entry of the first column is approximately 10000 bits long. Suppose first that we use the default variant.

```
> time C:=LLL(B:Proof:=false);
```

```
Time: 11.300
> R!(Norm (C[1]));
5.1959E121
```

The output basis is $(0.75, 0.501)$-reduced. Suppose now that we only wanted to have a basis that satisfies the main LLL properties with $\delta = 0.75$ and $\eta = 0.501$. Then we could have used the Siegel swapping condition.

```
> time C:=LLL(B:Proof:=false, SwapCondition:="Siegel");
Time: 10.740
> R!(Norm (C[1]));
6.6311E122
```

Notice that the quality of the output is quite often worse with the Siegel condition than with the Lovász condition, but the main LLL properties are satisfied anyway. Suppose now that we want a very well reduced basis. Then we fix δ and η close to 1 and 0.5 respectively.

```
> time C:=LLL(B:Proof:=false, Delta:=0.9999, Eta:=0.5001);
Time: 19.220
> R!(Norm (C[1]));
1.8056E121
```

This is of course more expensive, but the first output vector is significantly shorter. Finally, suppose that we only wanted to "shorten" the entries of the input basis, very efficiently and without any particular preference for the reduction factors δ and η. Then we could have used the `Fast` option, that tries to choose such parameters in order to optimize the running-time.

```
> time C:=LLL(B:Proof:=false, Fast:=1);
Time: 8.500
> R!(Norm (C[1]));
6.2746E121
```

By activating the verbose mode, one can know for which parameters the output basis is reduced.

```
> SetVerbose ("LLL", 1);
> C:=LLL(B:Proof:=false, Fast:=1);
[...]
The output basis is (0.830,0.670)-reduced
```

`LLL(X)`

`Al`	MonStgElt	*Default :* "*New*"
`Proof`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*FP*"
`Delta`	RngElt	*Default :* 0.75
`Eta`	RngElt	*Default :* 0.501
`InitialSort`	BoolElt	*Default :* `false`
`FinalSort`	BoolElt	*Default :* `false`

`StepLimit`	RngIntElt	*Default :* 0
`TimeLimit`	RngElt	*Default :* 0.0
`NormLimit`	RngIntElt	*Default :*
`UseGram`	BoolElt	*Default :* `false`
`DeepInsertions`	BoolElt	*Default :* `false`
`EarlyReduction`	BoolElt	*Default :* `false`
`SwapCondition`	MonStgElt	*Default :* "*Lovasz*"
`Fast`	RngIntElt	*Default :* 0
`Weight`	SeqEnum	*Default :* $[0, \dots, 0]$

Given a matrix X belonging to the matrix module $S = \mathrm{Hom}_R(M, N)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, compute a matrix Y whose non-zero rows are a LLL-reduced basis for the **Z**-lattice spanned by the rows of X (which need not be **Z**-linearly independent). The LLL function returns three values:

(a) A LLL-reduced matrix Y in S whose rows span the same lattice (**Z**-module) as that spanned by the rows of X.

(b) A unimodular matrix T in the matrix ring over **Z** whose degree is the number of rows of X such that $TX = Y$;

(c) The rank of X.

Note that the returned transformation matrix T is *always* over the ring of integers **Z**, even if R is not **Z**.

The input matrix X does not need to have linearly independent rows: this function performs a floating-point variant of what is usually known as the MLLL algorithm of M. Pohst [Poh87]. By default the rows of the returned matrix Y are sorted so that all the zero rows are at the bottom. The non-zero vectors are sorted so that they form a LLL-reduced basis (except when `FinalSort` is `true`; see below).

A detailed description of the parameters now follows. See the discussion and the example above this function for recommended parameters for common cases.

By default, the `Proof` option is set to `true`. It means that the result is guaranteed. It is possible, and usually faster, to switch off this option: it will perform the same calculations, without checking that the output is indeed reduced by using the L^2 algorithm. For the vast majority of cases, we recommend setting `Proof` to `false`.

By default the δ and η constants used in the LLL conditions are set respectively to 0.75 and 0.501 (except when `Method` is `"Integral"`, see below). The closer δ and η are to 1 and 0.5, respectively, the shorter the output basis will be. In the original description of the LLL algorithm, δ is 0.75 and η is 0.5. Making δ and η vary can have a significant influence on the running time. If `Method` is `"Integral"`, then by default η is set to 0.5 and δ to 0.99. With this value of `Method`, the parameter δ can be chosen arbitrarily in $(0.25, 1]$, but when δ is 1, the running time may increase dramatically. If `Method` is not `"Integral"`, then δ may be chosen arbitrarily in $(0.25, 1)$, and η may be chosen arbitrarily in $(0.5, \sqrt{\delta})$. Notice that the parameters δ and η used internally will be slightly larger and smaller than the given ones, respectively, to take floating-point inaccuracies into account and to ensure that the output basis will indeed be reduced for the expected parameters. In all cases, the output basis will be (δ, η)-LLL reduced.

For matrices over **Z** or **Q**, there are two main methods for handling the Gram-Schmidt orthogonalisation variables used in the algorithm: the Nguyen-Stehlé floating-point method (called L^2) and the exact integral method described by de Weger in [dW87]. The Nguyen-Stehlé algorithm is implemented as described in the article, along with a few faster heuristic variants. When `Method` is `"FP"`, these faster variants will be tried first, and when they are considered as misbehaving, they will be followed by the proved variant automatically. When `Method` is `"L2"`, the provable L^2 algorithm is used directly. Finally, when `Method` is `"Integral"`, the De Weger integral method is used. In this case, the `DeepInsertions`, `EarlyReduction`, `SiegelSwapCondition`, `NormLimit`, `Fast` and `Weight` options are unavailable (the code is older and has not been updated for these). Furthermore, the default value for `Eta` is then 0.5. The default `FP` method is nearly always *very* much faster than the other two methods.

By default, it is possible (though it happens very infrequently) that the returned basis is not of the expected quality. In order to be sure that the output is indeed correct, `Method` can be set to either `"Integral"` or `"L2"`.

The parameter `InitialSort` specifies whether the vectors should be sorted by length before the LLL reduction. When `FinalSort` is `true`, the vectors are sorted by increasing length (and alphabetical order in case of equality) after the LLL reduction (this parameter was called `Sort` before V2.13). The resulting vectors may therefore not be strictly LLL-reduced, because of the permutation of the rows.

The parameter `UseGram` specifies that for the floating-point methods (`L2` and `FP`) the computation should be performed by computing the Gram matrix F, using the `LLLGram` algorithm below, and by updating the basis matrix correspondingly. Magma will automatically do this internally for the floating-point method if it deems that it is more efficient to do so, so this parameter just allows one to stipulate that this method should or should not be used. For the integral method, using the Gram matrix never improves the computation, so the value of this parameter is simply ignored in this case.

Setting the parameter `StepLimit` to a positive integer s will cause the function to terminate after s steps of the main loop. Setting the parameter `TimeLimit` to a positive real number t will cause the function to terminate after the algorithm has been executed for t seconds (process user) time. Similarly, setting the parameter `NormLimit` to a non-negative integer N will cause the algorithm to terminate after a vector of norm less or equal to N has been found. In such cases, the current reduced basis is returned, together with the appropriate transformation matrix and an upper bound for the rank of the matrix. Nothing precise can then be said exactly about the reduced basis (it will not be LLL-reduced in general of course) but will at least be reduced to a certain extent.

When the value of the parameter `DeepInsertions` is `true`, Schnorr-Euchner's deep insertion algorithm is used [SE94]. It usually provides better bases, but can take significantly more time. In practice, one may first LLL-reduce the basis and then use the deep insertion algorithm on the computed basis.

When the parameter `EarlyReduction` is `true` and when any of the two floating-point methods are used, some early reduction steps are inserted inside the execution of the LLL algorithm. This sometimes makes the entries of the basis smaller very quickly. It occurs in particular for lattice bases built from minimal polynomial or integer relation detection problems. The speed-up is sometimes dramatic, especially if the reduction in length makes it possible to use C integers for the basis matrix (instead of multiprecision integers) or C doubles for the floating-point calculations (instead of multiprecision floating-point numbers or doubles with additional exponent).

The parameter `SwapCondition` can be set either to `"Lovasz"` (default) or to `"Siegel"`. When its value is `"Lovasz"`, the classical Lovász swapping condition is used, whereas otherwise the so-called *Siegel* condition is used. The Lovász condition tests whether the inequality $\|b_i^* + \mu_{i,i-1} b_{i-1}^*\|^2 \geq \delta \|b_{i-1}^*\|^2$ is satisfied or not, whereas in the Siegel case the inequality $\|b_i^*\|^2 \geq (\delta - \eta^2)\|b_{i-1}^*\|^2$ is used (see [Akh02] for more details). In the Siegel case, the execution may be significantly faster. Though the output basis may not be LLL-reduced, the classical LLL inequalities (see the introduction of this subsection) will be fulfilled.

When the option `Fast` is set to 1 or 2, all the other parameters may be changed internally to end the execution as fast as possible. Even if the other parameters are not set to the default values, they may be ignored. This includes the parameters `Delta`, `Eta` and `SwapCondition`, so that the output basis may not be LLL-reduced. However, if the verbose mode is activated, the chosen values of these parameters will be printed, so that the classical LLL inequalities (see the introduction of this subsection) may be used afterwards. When `Fast` is set to 2, the chosen parameters are such that the classical LLL inequalities will be at least as strong as for the default parameters `Delta` and`Eta`.

If `Weight` is the list of integers $[x_1, \ldots, x_n]$ (where n is the degree of the lattice), then the LLL algorithm uses a weighted Euclidean inner product: the inner product of the vectors $(v_1, v_2, \ldots, v_n)$ and $(w_1, w_2, \ldots, w_n)$ is defined to be $\sum_{i=1}^{n} \left(v_i \cdot w_i \cdot 2^{2x_i}\right)$.

By default, the `Al` parameter is set to `New`. If it is set to `Old`, then the former Magma LLL is used (prior to V2.13); notice that in this case, the execution is not guaranteed to finish and that the output basis may not be reduced for the given LLL parameters. This variant is not maintained anymore.

Note 1: If the input basis has many zero entries, then try to place them in the last columns; this will slightly improve the running time.
Note 2: If the elementary divisors of the input matrix are fairly small, relative to the size of the matrix and the size of its entries, then it may be very much quicker

to reduce the matrix to Hermite form first (using the function `HermiteForm`) and then to LLL-reduce this matrix. This case can arise: for example, when one has a "very messy" matrix for which it is known that the lattice described by it has a relatively short basis.
Note 3: Sometimes, the `EarlyReduction` variant can significantly decrease the running time.

`BasisReduction(X)`

`BasisReduction(X)`

This is a shortcut for `LLL(X:Proof:=false)`.

`LLLGram(F)`

`Isotropic` BoolElt *Default :* `false`

Given a symmetric matrix F belonging to the the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to a LLL-reduced form of the matrix X. The rows of the corresponding generator matrix X need not be **Z**-linearly independent in which case F will be singular. This function returns three values:

(a) A LLL-reduced Gram matrix G in S of the Gram matrix F;

(b) A unimodular matrix T in the matrix ring over **Z** whose degree is the number of rows of F such that $G = TFT^{tr}$.

(c) The rank of F (which equals the dimension of the lattice generated by X).

The options available are the same as for the `LLL` routine, except of course the `UseGram` option that has no sense here. The `Weight` parameter should not be used either. The routine can be used with any symmetric matrix (possibly not definite nor positive): Simon's indefinite LLL variant (which puts absolute values on the Lovász condition) is used (see [Sim05]).

When the option `Isotropic` is true, some attempt to deal with isotropic vectors is made. (This is only relevant when working on an indefinite Gram matrix). In particular, if the determinant is squarefree, hyperbolic planes are split off iteratively.

`LLLBasisMatrix(L)`

Given a lattice L with basis matrix B, return *the LLL basis matrix* B' of L, together with the transformation matrix T such that $B' = TB$. The LLL basis matrix B' is simply defined to be a LLL-reduced form of B; it is stored in L when computed and subsequently used internally by many lattice functions. The LLL basis matrix will be created automatically internally as needed with $\delta = 0.999$ by default (note that this is different from the usual default of 0.75); by the use of parameters to this function one can ensure that the LLL basis matrix is created in a way which is different to the default. The parameters (not listed here again) are the same as for the function `LLL` (q.v.), except that the limit parameters are illegal.

```
LLLGramMatrix(L)
```

Given a lattice L with Gram matrix F, return the *LLL Gram matrix* F' of L, together with the transformation matrix T such that $F' = TFT^{tr}$. F' is simply defined to be $B'(B')^{tr}$, where B' is the LLL basis matrix of L—see the function `LLLBasisMatrix`. The parameters (not listed here again) are the same as for the function `LLL` (q.v.), except that the limit parameters are illegal.

```
LLL(L)
```

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is LLL-reduced and $B' = TB$. Note that the inner product used in the LLL computation is that given by the inner product matrix of L (so, for example, the resulting basis may not be LLL-reduced with respect to the standard Euclidean norm). The LLL basis matrix of L is used, so calling this function with argument L is completely equivalent (ignoring the second return value) to the invocation `LatticeWithBasis(LLLBasisMatrix(L), InnerProductMatrix(L))`. The parameters (not listed here again) are the same as for the function `LLL` (q.v.), except that the limit parameters are illegal. The `BasisReduction` shortcut to turn off the `Proof` option is also available.

```
BasisReduction(L)
```

This is a shortcut for `LLL(L:Proof:=false)`.

```
SetVerbose("LLL", v)
```

(Procedure.) Set the verbose printing level for the LLL algorithm to be v. Currently the legal values for v are `true`, `false`, 0, 1, 2 and 3 (`false` is the same as 0, and `true` is the same as 1). The three non-zero levels notify when the maximum LLL-reduced rank of the LLL algorithm increases, level 2 also prints the norms of the reduced vectors at such points, and level 3 additionally gives some current status information every 15 seconds.

Example H30E8

We demonstrate how the LLL algorithm can be used to find very good multipliers for the extended GCD of a sequence of integers. Given a sequence $Q = [x_1, \ldots, x_n]$ of integers we wish to find the GCD g of Q and integers m_i for $1 \le i \le m$ such that $g = m_1 \cdot x_1 + \ldots + m_n \cdot x_m$.
For this example we set Q to a sequence of $n = 10$ integers of varying bit lengths (to make the problem a bit harder).

```
> Q := [ 67015143, 248934363018, 109210, 25590011055, 74631449,
>        10230248, 709487, 68965012139, 972065, 864972271 ];
> n := #Q;
> n;
10
```

We next choose a scale factor S large enough and then create the $n \times (n+1)$ matrix $X = [I_n|C]$ where C is the column vector whose i-th entry is $S \cdot Q[i]$.

```
> S := 100;
> X := RMatrixSpace(IntegerRing(), n, n + 1) ! 0;
> for i := 1 to n do X[i][i + 1] := 1; end for;
> for i := 1 to n do X[i][1] := S * Q[i]; end for;
> X;
[6701514300     1 0 0 0 0 0 0 0 0 0]
[24893436301800 0 1 0 0 0 0 0 0 0 0]
[10921000       0 0 1 0 0 0 0 0 0 0]
[2559001105500  0 0 0 1 0 0 0 0 0 0]
[7463144900     0 0 0 0 1 0 0 0 0 0]
[1023024800     0 0 0 0 0 1 0 0 0 0]
[70948700       0 0 0 0 0 0 1 0 0 0]
[6896501213900  0 0 0 0 0 0 0 1 0 0]
[97206500       0 0 0 0 0 0 0 0 1 0]
[86497227100    0 0 0 0 0 0 0 0 0 1]
```

Finally, we compute a LLL-reduced form L of X.

```
> L := LLL(X);
> L;
[   0    0    1    0  -15   -6    3    1    2   -3   -3]
[   0   -3    5   -3  -11   -1    0    8  -14    4    3]
[   0   -5   -2    2   -2   14    8   -6    8    1   -4]
[   0    6    1   -3  -10   -3  -14    5    0    8    8]
[   0   -1   -2    0   -2  -13    8    6    8   10   -2]
[   0   -9   -3  -11    5    7   -1    1    9  -11   -2]
[   0   16    0    3    3    9   -3    0   -1   -4  -11]
[   0   -6    1  -16    4   -1    6   -6   -5    7   -7]
[   0    9   -3  -10    7   -3    1   -7    8    0   18]
[-100   -3   -1   13   -1   -4    2    3    4    5   -1]
```

Notice that the large weighting on the first column forces the LLL algorithm to produce as many vectors as possible at the top of the matrix with a zero entry in the first column (since such vectors will have shorter norms than any vector with a non-zero entry in the first column). Thus the GCD of the entries in Q is the entry in the bottom left corner of L divided by S, viz. 1. The last n entries of the last row of L gives a sequence of multipliers M for the extended GCD algorithm. Also, taking the last n entries of each of the first $n-1$ rows of L gives independent null-relations for the entries of Q (i.e., the kernel of the corresponding column matrix). We check that M gives a correct sequence of multipliers.

```
> M := Eltseq(L[10])[2 .. n+1]; M;
[ 3, 1, -13, 1, 4, -2, -3, -4, -5, 1 ]
> &+[Q[i]*M[i]: i in [1 .. n]];
-1
```

30.7.2 Pair Reduction

`PairReduce(X)`

Given a matrix X belonging to the the matrix module $S = \mathrm{Hom}_R(M, N)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbf{Z}$ or $\mathbf{Q}$, compute a matrix Y whose rows form a pairwise reduced basis for the $\mathbf{Z}$-lattice spanned by the rows of X. Being pairwise reduced (i.e., $2|(v, w)| \le \min(\|v\|, \|w\|)$ for all pairs of basis vectors) is a much simpler criterion than being LLL-reduced, but often yields sufficiently good results very quickly. It can also be used as a preprocessing for LLL or in alternation with LLL to obtain better bases. The rows of X need not be Z-linearly independent. This function returns two values:

(a) The pairwise reduced matrix Y row-equivalent to X as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbf{Z}$ whose degree is the number of rows of X such that $TX = Y$.

`PairReduceGram(F)`

`Check` Bool<span style="font-variant:small-caps">Elt</span> *Default :* `false`

Given a symmetric positive semidefinite matrix F belonging to the the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbf{Z}$ or $\mathbf{Q}$, so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to a pairwise reduced form of the matrix X. The rows of the corresponding matrix X need not be Z-linearly independent. This function returns two values:

(a) The pairwise reduced Gram matrix G of F as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbf{Z}$ whose degree is the number of rows of F which gives the corresponding transformation: $G = TFT^{tr}$.

The matrix F must be a symmetric positive semidefinite matrix; if it is not the results are unpredictable. By default, Magma does not check this since it may be expensive in higher dimensions and in many applications will be known a priori. The checking can be invoked by setting `Check := true`.

`PairReduce(L)`

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is pairwise reduced and $B' = TB$. Note that the inner product used in the pairwise reduction computation is that given by the inner product matrix of L (so, for example, the resulting basis may not be pairwise reduced with respect to the standard Euclidean norm).

30.7.3 Seysen Reduction

```
Seysen(X)
```

Given a matrix X belonging to the the matrix module $S = \mathrm{Hom}_R(M, N)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbf{Z}$ or $\mathbf{Q}$, compute a matrix Y whose rows form a Seysen-reduced basis for the $\mathbf{Z}$-lattice spanned by the rows of X. The rows of X need not be Z-linearly independent. The Seysen-reduced matrix Y is such that the entries of the corresponding Gram matrix $G = YY^{tr}$ and its inverse G^{-1} are simultaneously reduced. This function returns two values:

(a) The Seysen-reduced matrix Y corresponding to X as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbf{Z}$ whose degree is the number of rows of X such that $TX = Y$.

```
SeysenGram(F)
```

`Check` BoolElt *Default :* `false`

Given a symmetric positive semidefinite matrix F belonging to the the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbf{Z}$ or $\mathbf{Q}$, so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to a Seysen-reduced form of the matrix X. The rows of the corresponding matrix X need not be Z-linearly independent. The Seysen-reduced Gram matrix G is such that the entries of G and its inverse G^{-1} are simultaneously reduced. This function returns two values:

(a) The Seysen-reduced Gram matrix G of F as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbf{Z}$ whose degree is the number of rows of F which gives the corresponding transformation: $G = TFT^{tr}$.

The matrix F must be a symmetric positive semidefinite matrix; if it is not the results are unpredictable. By default, Magma does not check this since it may be expensive in higher dimensions and in many applications will be known a priori. The checking can be invoked by setting `Check := true`.

```
Seysen(L)
```

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is Seysen-reduced and $B' = TB$. Note that the inner product used in the Seysen-reduction computation is that given by the inner product matrix of L (so, for example, the resulting basis may not be Seysen-reduced with respect to the standard Euclidean norm). The effect of the reduction is that the basis of L' and the dual basis of L' will be simultaneously reduced.

Example H30E9

We demonstrate how the function `Seysen` can be used on the Gram matrix of the Leech lattice to obtain a gram matrix S for the lattice so that both S and S^{-1} are simultaneously reduced. Note that all three reduction methods yield a basis of vectors of minimal length, but the Seysen reduction also has a dual basis of vectors of norm 4. This is of interest in representation theory, for example, as the entries of the inverse of the Gram matrix control the size of the entries in a representation on the lattice.

```
> F := GramMatrix(Lattice("Lambda", 24));
> [ F[i][i] : i in [1..24] ];
[ 8, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ (F^-1)[i][i] : i in [1..24] ];
[ 72, 8, 12, 8, 10, 4, 4, 8, 8, 4, 4, 8, 4, 4, 4, 4, 4, 4, 4, 4, 8, 4, 4, 8 ]
> L := LLLGram(F);
> P := PairReduceGram(F);
> S := SeysenGram(F);
> [ L[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ P[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ S[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ (L^-1)[i][i] : i in [1..24] ];
[ 6, 4, 8, 4, 32, 4, 4, 8, 18, 4, 4, 8, 8, 8, 12, 4, 6, 4, 20, 4, 8, 4, 14, 4 ]
> [ (P^-1)[i][i] : i in [1..24] ];
[ 72, 40, 12, 8, 10, 4, 4, 8, 8, 4, 4, 8, 4, 4, 4, 4, 4, 4, 4, 4, 8, 4, 4, 8 ]
> [ (S^-1)[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
```

30.7.4 HKZ Reduction

An ordered set of d linearly independent vectors $b_1, b_2, \ldots, b_d \in \mathbf{R}^n$ is said to be Hermite-Korkine-Zolotarev-reduced (HKZ-reduced for short) if the three following conditions are satisfied:

(a) For any $i > j$, we have $|\mu_{i,j}| \leq 0.501$,

(b) The vector b_1 is a shortest non-zero vector in the lattice spanned by $b_1, b_2, \ldots, b_d$,

(c) The vectors $b_2 - \mu_{2,1}b_1, \ldots, b_d - \mu_{d,1}b_1$ are themselves HKZ-reduced,

where $\mu_{i,j} = (b_i, b_j^*)/\|b_j^*\|^2$ and b_i^* is the i-th vector of the Gram-Schmidt orthogonalisation of $(b_1, b_2, \ldots, b_d)$.

`HKZ(X)`

`Proof`	BoolElt	*Default* : `true`
`Unique`	BoolElt	*Default* : `false`
`Prune`	SeqEnum	*Default* : $[\ldots,[1.0,\ldots,1.0],\ldots]$

Given a matrix X over a subring of the real field, compute a matrix Y whose nonzero rows are an HKZ-reduced basis for the **Z**-lattice spanned by the rows of X (which need not be **Z**-linearly independent). The HKZ function returns an HKZ-reduced matrix Y whose rows span the same lattice (**Z**-module) as that spanned by the rows of X, and a unimodular matrix T in the matrix ring over **Z** whose degree is the number of rows of X such that $TX = Y$.

Although the implementation relies on floating-point arithmetic, the result can be guaranteed to be correct. By default, the `Proof` option is set to `true`, thus guaranteeing the output. The run-time might be improved by switching off this option.

A given lattice may have many HKZ-reduced bases. If the `Unique` option is turned on, a uniquely determined HKZ-reduced basis is computed. This basis is chosen so that: for any $i > j$, we have $\mu_{i,j} \in [-1/2, 1/2)$; for any i, the first nonzero coordinate of the vector b_i^* is positive; and for any i, the vector b_i^* is the shortest among possible vectors for the lexicographical order.

The implementation solves the Shortest Vector Problem for lattices of dimensions 1 to d, where d is the number of rows of the input matrix X. The latter instances of the Shortest Vector Problem are solved with the same enumeration algorithm as is used for computing the minimum of a lattice, which can be expressed as a search within a large tree. The i-th table of the `Prune` optional parameter is used to prune the i-dimensional enumeration. The default value is:

$[[1.0 : j \texttt{ in } [1..i]] : i \texttt{ in } [1..\texttt{NumberOfRows}(X)]]$.

See the introduction of Section 30.8 for more details on the `Prune` option.

`HKZGram(F)`

`Proof`	BoolElt	*Default* : `true`
`Prune`	SeqEnum	*Default* : $[\ldots,[1.0,\ldots,1.0],\ldots]$

Given a symmetric positive semidefinite matrix F over a subring of the real field so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to an HKZ-reduced form of the matrix X. The function returns the HKZ-reduced Gram matrix G of F, and a unimodular matrix T in the matrix ring over **Z** whose degree is the number of rows of F which gives the corresponding transformation: $G = TFT^{tr}$.

`HKZ(L)`

Proof	BoolElt	*Default :* `true`
Prune	SeqEnum	*Default :* $[\ldots,[1.0,\ldots,1.0],\ldots]$

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is HKZ-reduced and $B' = TB$.

`SetVerbose("HKZ", v)`

(Procedure.) Set the verbose printing level for the HKZ algorithm to be v. Currently the legal values for v are `true`, `false`, 0 and 1 (`false` is the same as 0, and `true` is the same as 1). More information on the progress of the computation can be obtained by setting the `"Enum"` verbose on.

`GaussReduce(X)`

`GaussReduceGram(F)`

`GaussReduce(L)`

Restrictions of the `HKZ` functions to lattices of rank 2.

Example H30E10

HKZ-reduced bases are much harder to compute than LLL-reduced bases, but provide a significantly more powerful representation of the spanned lattice. For example, computing all short lattice vectors is more efficient if one starts from an HKZ-reduced basis.

```
> d:=60;
> B:=RMatrixSpace(IntegerRing(), d, d)!0;
> for i:=1 to d do for j:=1 to d do B[i][j]:=Random(100*d); end for; end for;
> time C1 := LLL(B);
Time: 0.020
> time C2 := HKZ(B);
Time: 1.380
> m := Norm(C2[1]);
> time _:=ShortVectors(Lattice(C1), 11/10*m);
Time: 1.750
> time _:=ShortVectors(Lattice(C2), 11/10*m);
Time: 0.850
> time _:=ShortVectors(Lattice(C1), 3/2*m);
Time: 73.800
> time _:=ShortVectors(Lattice(C2), 3/2*m);
Time: 32.220
```

30.7.5 Recovering a Short Basis from Short Lattice Vectors

```
ReconstructLatticeBasis(S, B)
```

Given a basis S of a finite index sublattice of the lattice L spanned by the rows of the integral matrix B, return a matrix C whose rows span L and are not much longer than those of S. Specifically, the algorithm described in Lemma 7.1 of [MG02] is implemented, and the rows of the output matrix satisfy the following properties:

(a) For any i, $\|c_i\| \leq i^{1/2}\|s_i\|$,

(b) For any i, $\|c_i^*\| \leq \|s_i^*\|$,

where c_i^* (respectively s_i^*) denotes the i-th vector of the Gram-Schmidt orthogonalisation of $(c_1, c_2, \ldots, c_d)$ (respectively $(s_1, s_2, \ldots, s_d)$) .

30.8 Minima and Element Enumeration

The functions in this section are all based on one algorithm which enumerates all vectors of a lattice in a specified hyperball [FP83, Kan83, SE94]. As the underlying computational problems (the Shortest and Closest Lattice Vector Problems) are hard [Ajt98, vEB81], the general application of this algorithm is restricted to lattices of moderate dimension (up to 50 or 60). However, some tasks like finding a couple of short vectors or finding the minimum of a lattice without symmetry may still be feasible in higher dimensions. The function `EnumerationCost` provides an estimate of the cost of running the enumeration algorithm on a specified input. This allows to know beforehand if the computation is likely to terminate within a reasonable amount of time.

For each function which enumerates short vectors of a lattice L, there is a corresponding function which enumerates vectors of L which are close to a given vector w (which usually lies outside L). Note that if one wishes to enumerate short vectors of the coset $L+w$ of the lattice L, where w is any vector of degree compatible with L, one can simply enumerate vectors $v \in L$ close to $-w$ and then just take the vectors $v + w$ for each v as the short vectors of the coset.

The enumeration routine underlying all the functions described below relies on floating-point approximations. However, it can be run in a rigorous way in some cases, see [PS08]. By default, the outputs of the functions `Minimum`, `PackingRadius`, `HermiteNumber`, `CentreDensity`, `Density`, `KissingNumber`, `ShortestVectors`, `ShortestVectorsMatrix`, `ShortVectors`, `ShortVectorsMatrix` and `ThetaSeries` are guaranteed to be correct. This correctness guarantee can be turned off by setting the optional parameter `Proof` to `false`. However, this comes virtually for free, the additional computations being most often negligible. In the present version of Magma, the outputs of all other functions are only likely to be correct.

The enumeration algorithm from [FP83, Kan83, SE94] can be interpreted as a search within a large tree. This can be extremely time-consuming. Schnorr, Euchner and Hörner [SE94, SH95] introduced techniques to prune the latter tree: the correct output might be missed, but the execution of the algorithm is likely to terminate faster. A new pruning strategy is available for the functions `Minimum`, `PackingRadius`, `HermiteNumber`, `CentreDensity`, `Density`, `KissingNumber`, `ShortestVectors`, `ShortestVectorsMatrix`, `ShortVectors`, `ShortVectorsMatrix` and `ThetaSeries`. Naturally, if the pruning strategy is used, the result cannot be guaranteed to be correct anymore. Let $(b_1, b_2, \ldots, b_d)$ be a basis of a lattice L, let $(b_1^*, b_2^*, \ldots, b_d^*)$ denote its Gram-Schmidt orthogonalisation, and let $\mu_{i,j} = (b_i, b_j^*)/\|b_j^*\|^2$ for $i \geq j$. Suppose we are interested in finding all vectors in L of norm $\leq u$. The enumeration algorithm considers the equations:

$$\| \textstyle\sum_{j=i}^{d} \left(\sum_{k=j}^{d} \mu_{k,j} x_k \right) b_j^* \|^2 \leq u, \text{ for } i = d, d-1, \ldots, 1,$$

where the x_k's are integers. If the `Prune` optional parameter is set to $[p_1, \ldots, p_d]$, then the equations above will be replaced by:

$$\| \textstyle\sum_{j=i}^{d} \left(\sum_{k=j}^{d} \mu_{k,j} x_k \right) b_j^* \|^2 \leq p_i u, \text{ for } i = d, d-1, \ldots, 1.$$

The p_i's must belong to the interval $[0, 1]$. For a given input $((b_1, \ldots, b_d), u)$ to the enumeration procedure, it is possible to heuristically estimate both the running-time gain and the probability of missing a solution. See Subsection 30.8.6 for more details.

30.8.1 Minimum, Density and Kissing Number

The functions in this subsection compute invariants of a lattice which are all related to its minimum. See [JC98] for background about minimum, density and kissing numbers.

`Minimum(L)`

`Min(L)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the minimum of the lattice L, i.e., the minimal norm of a non-zero vector in the lattice. Note that this is in general a hard problem and may be very time consuming. See also the attributes section below for how to assert the minimum of a lattice.

`PackingRadius(L)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

The packing radius is half the square root of the minimum of the non-zero lattice L.

`HermiteConstant(n)`

Return the n-th Hermite constant raised to the power of n. The exact value is provided if $n \leq 8$ or $n = 24$, and otherwise an upper bound is returned. The n-th Hermite constant is defined as the maximum of `Min`(L)/`Determinant`(L) over all n-dimensional lattices L.

`HermiteNumber(L)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the Hermite number of the non-zero lattice L, i.e., `Min`(L)/`Determinant`(L).

`CentreDensity(L)`

`CenterDensity(L)`

`CentreDensity(L, K)`

`CenterDensity(L, K)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

The center density of the lattice L, as an element of the real field K. This is defined to be the square root of (`Min`(L)/4)$^{\texttt{Rank(L)}}$/`Determinant(L)`. The argument for the real field K may be omitted, in which case K is taken to be the current default real field.

The product of the centre density by the volume of a sphere of radius 1 in n-dimensional space gives the density of the lattice-centered sphere packing of L, called the density of the lattice.

`Density(L)`

`Density(L, K)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

The density of the lattice L, as an element of the real field K, i.e., the density of the lattice-centered sphere packing. If the argument for the real field K is omitted, the field K is taken to be the default real field.

`KissingNumber(L)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the kissing number of the lattice L, which equals the number of vectors of minimal non-zero norm (twice the length of the sequence returned by `ShortestVectors(L)` since that returns only normalized vectors). This is the maximum number of nonoverlapping spheres of diameter `Minimum(L)` with centres at lattice points which touch a fixed sphere of the same diameter with centre at a lattice point.

Example H30E11

We create the Leech lattice Λ_{24} and compute its minimum (4), density, and kissing number (196560). Note that the computation of the minimum is very fast since the lattice is even and at least one basis element has norm 4; one has only to prove that there are no vectors of norm 2.

```
> L := Lattice("Lambda", 24);
> IsEven(L), Norm(L.2);
true 4
> time Minimum(L);
4
Time: 0.020
> Density(L);
0.001929574309403923047903345563 69
> time KissingNumber(L);
196560
Time: 0.180
```

30.8.2 Shortest and Closest Vectors

`ShortestVectors(L)`

`Max`	RngIntElt	*Default :* ∞
`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the shortest non-zero vectors of the lattice L, i.e., the vectors $v \in L$ such that (v, v) is minimal, as a sorted sequence Q. The vectors are computed up to sign (so only one of v and $-v$ appears in Q) and normalized so that the first non-zero entry in each vector is positive, and Q is sorted with respect to lexicographic order. By default, all the (normalized) vectors are computed; the optional parameter `Max` allows the user to specify the maximal number of computed vectors. Note that unless the minimum of the lattice is already known, it has to be computed by this function, which may be very time consuming.

`ShortestVectorsMatrix(L)`

`Max`	RngIntElt	*Default :* ∞
`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the shortest non-zero vectors of the lattice L as the rows of a matrix. This is more efficient or convenient for some applications than forming the sequence of vectors. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space). By default, all the (normalized) vectors are computed; the optional parameter `Max` allows the user to specify the maximal number of computed vectors.

`ClosestVectors(L, w)`

`Max` RngIntElt *Default :* ∞

Return the vectors of the lattice L which are closest to the vector w, together with the minimal squared distance d. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. The closest vectors are those vectors $v \in L$ such that the squared distance $(v - w, v - w)$ between v and w (and thus the distance between v and w) is minimal; this minimal squared distance is the second return value d. The vectors are returned as a sequence Q, sorted with respect to lexicographic order. Note that the closest vectors are *not* symmetrical with respect to sign (while the shortest vectors are) so the returned closest vectors are not normalized. By default, all the closest vectors are computed; the optional parameter `Max` allows the user to specify the maximal number of computed vectors.

`ClosestVectorsMatrix(L, w)`

`Max` RngIntElt *Default :* ∞

Return the vectors of the lattice L which are closest to the vector w as a matrix, together with the squared distance d. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space). The vectors are returned as sequence Q, sorted with respect to lexicographic order. Note that the closest vectors are *not* symmetrical with respect to sign (while the shortest vectors are) so the returned closest vectors are not normalized. By default, all the closest vectors are computed; the optional parameter `Max` allows the user to specify the maximal number of computed vectors.

Example H30E12

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 $[2, 2, 2, 2, 2, 2, 2, 2]$ (see the function `SuccessiveMinima` below).

```
> 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 an actual 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
```

30.8.3 Short and Close Vectors

`ShortVectors(L, u)`

`ShortVectors(L, l, u)`

`Max`	RngIntElt	*Default :* ∞
`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \dots, 1.0]$

Return the vectors of the lattice L with norm within the prescribed range, together with their norms, as a sorted sequence Q. The sequence Q contains tuples of the form $< v, r >$ where v is a vector and r its norm. Either one positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. The vectors are computed up to sign (so only one of v and $-v$ appears in Q) and normalized so that the first non-zero entry in each vector is positive, and Q is sorted with respect to first the norms and then the lexicographic order for the vectors. By default, all the (normalized) vectors with norm in the prescribed range are computed. The optional parameter `Max` allows the user to specify the maximal number of computed vectors.

```
ShortVectorsMatrix(L, u)
ShortVectorsMatrix(L, l, u)
```

`Max`	RngIntElt	*Default :* ∞
`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

This is very similar to `ShortVectors`, but returns the vectors of the lattice L with norm within the prescribed range as the rows of a matrix S rather than as a sequence of tuples. This is more efficient or convenient for some applications than forming the sequence. By default, all the (normalized) vectors with norm in the prescribed range are computed. The optional parameter `Max` allows the user to specify the maximal number of computed vectors. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space).

```
CloseVectors(L, w, u)
CloseVectors(L, w, l, u)
```

`Max`	RngIntElt	*Default :* ∞

Return the vectors of the lattice L whose squared distance from the vector w is within the prescribed range, together with their squared distances, as a sorted sequence Q. The returned sequence Q contains tuples of the form $< v, d >$ where v is a vector from L and $d = (v - w, v - w)$ is its squared distance from w. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. Either one positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. Note that close vectors are *not* symmetrical with respect to sign (while short vectors are) so the returned close vectors are not normalized. Q is sorted with respect to first the squared distances and then the lexicographic order for the vectors. By default, all the vectors with squared distance in the prescribed range are computed. The optional parameter `Max` allows the user to specify the maximal number of computed vectors.

`CloseVectorsMatrix(L, w, u)`

`CloseVectorsMatrix(L, w, l, u)`

`Max` RngIntElt *Default :* ∞

This is very similar to `CloseVectors`, but returns the vectors of the lattice L whose squared distance from the vector w is within the prescribed range as the rows of a matrix C rather than as a sequence of tuples. This is more efficient or convenient for some applications than forming the sequence. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. By default, all the close vectors are computed. The optional parameter `Max` allows the user to specify the maximal number of computed vectors. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space).

Example H30E13

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 (op. cit., at the beginning of this chapter). 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
Determinant: 1846735827632128
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)
( 1  1  1  1 -3  1  1 -1 -1  1 -1 -1  0  1)
( 1 -1 -3  1  1 -1 -1  1 -1  1  1  1  0  1)
( 2  2  0  0  0 -2  2  2 -2  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  3 -1 -1 -1 -1  1  0 -1)
(-1 -1 -1 -1  1  1  1  1 -1 -1 -1  1  0  1)
( 2  0  0  2  0  2  0  0  0  0 -2  0  0 -2)
(-1 -1 -1  1  1 -1 -1  1 -1  1  1 -3  0 -1)
(-1 -1  1 -1 -1  1 -1 -1 -1  3 -1  1  0 -3)
( 0 -2  0  0  2  0 -2  0 -2  0  0  0 12  0)
> 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 $\{1 \dots 50\}$ 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, 3, 4, 7, 10, 11, 13, 14, 18, 20, 22, 23, 26, 28, 29, 34, 35, 37, 40, 41,
42, 45, 48, 49, 50 ]
> s := &+[Q[i]: i in I]; Ilog2(s);
1003
> time L := KnapsackLattice(Q, s);
Time: 0.570
> [Ilog2(Norm(b)): b in Basis(L)];
[ 5, 46, 46, 45, 47, 47, 46, 46, 46, 46, 46, 46, 46, 45, 47, 46, 46, 46, 46,
47,46, 46, 45, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 45, 46, 46,
45, 45, 46, 46, 46, 46, 46, 45, 46, 46, 46, 46, 46 ]
> time KnapsackSolutions(L);
[
    [ 1, 3, 4, 7, 10, 11, 13, 14, 18, 20, 22, 23, 26, 28, 29, 34, 35, 37, 40,
    41, 42, 45, 48, 49, 50 ]
]
Time: 0.040
```

Example H30E14

In this example we demonstrate how short vectors of lattices can be used to split homogeneous components of integral representations. We first define an integral matrix group of degree 8. The group is isomorphic to A_5 and the representation contains the 4-dimensional irreducible representation of A_5 with multiplicity 2.

```
> G := MatrixGroup< 8, Integers() |
>   [ -673,  -291,  -225,  -316,   250,   -32,    70,  -100,
>      252,   274,   349,   272,  -156,   -94,  -296,   218,
>     2532,  1159,   609,  5164, -1450,  -181,   188,   742,
>     -551,   163,   629, -1763,   285,  -162,  -873,   219,
>    -2701,  -492,   411, -3182,  1062,  -397, -1195,   151,
>    -5018, -1112,  1044,-12958,  2898,  -153, -2870,  -454,
>     2581,    90, -1490,  8197, -1553,   261,  2556,  -149,
>    -3495, -2776, -3218, -2776,  1773,   652,  2459, -1910],
>   [ 2615,  1314,  1633,   400,  -950, -1000, -2480,  1049,
>      161,   159,   347,  -657,     2,  -385,  -889,   230,
>    -2445, -1062, -1147,   269,   744,  1075,  2441,  -795,
>     1591,   925,  1454, -1851,  -350, -1525, -3498,   982,
>    10655,  5587,  7476, -1751, -3514, -5575,-13389,  4873,
>     6271,  3253,  4653, -6126, -1390, -5274,-11904,  3219,
>    -3058, -1749, -2860,  5627,   392,  3730,  8322, -2009,
>     4875,  1851,  1170,  5989, -2239,   625,  1031,   692] >;
> Order(G);
60
```

Since the group is small enough we can generate elements in the endomorphism ring by averaging over the group elements.

```
> M := MatrixRing(Integers(), 8);
> e := [ &+[ M!g * MatrixUnit(M, i, i) * M!(g^-1) : g in G ] : i in [1..4] ];
> E := sub<M | e>;
> Dimension(E);
4
```

We now transform E into a lattice of dimension 4 and degree 64 and rescale the basis vectors so that they have lengths of the same order of magnitude.

```
> L := Lattice(64, &cat[ Eltseq(b) : b in Basis(E) ]);
> LL := sub<L | [ Round( Norm(L.4)/Norm(L.i) ) * L.i : i in [1..4] ]>;
> Minimum(LL);
910870284600
> SV := ShortVectors(LL, 100*Minimum(LL));
> #SV;
46
> Sing := [ X : v in SV | Determinant(X) eq 0 where X is M!Eltseq(v[1]) ];
> #Sing;
3
```

We thus have found three singular elements amongst the 46 shortest vectors and use the kernel of the first of these to get the representation on a subspace of dimension 4.

```
> ker := LLL( KernelMatrix(Sing[1]) );
> ker;
[ 10   0  -9   5  -2   0   5  -4]
[ -3  -4   1 -16   3   5   4  -3]
[ -8 -11 -10  -1   4   0   6  -5]
[-13   3   4  10   1   5   8   2]
> H := MatrixGroup<4, Integers() | [Solution(ker, ker*g) : g in Generators(G)]>;
> H;
MatrixGroup(4, Integer Ring)
Generators:
    [ 1  0  0  0]
    [-3 -2 -4 -5]
    [-2 -3 -3 -4]
    [ 2  3  4  5]
    [-1 -1 -1 -1]
    [ 2  1  2  2]
    [-4 -2 -1 -2]
    [ 3  2  1  2]
> #H;
60
```

30.8.4 Short and Close Vector Processes

`ShortVectorsProcess(L, u)`

`ShortVectorsProcess(L, l, u)`

Given a lattice L and a range, create a corresponding short vectors lattice enumeration process P. This process provides the environment for enumerating each vector $v \in L$ with norm within the range. Either a positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. Successive calls to `NextVector` (see below) will result in the enumeration of the vectors.

`CloseVectorsProcess(L, w, u)`

`CloseVectorsProcess(L, w, l, u)`

Given a lattice L, a vector w and a range, create a corresponding close vectors lattice enumeration process P. This process provides the environment for enumerating each vector $v \in L$ such its squared distance $(v - w, v - w)$ from w is within the range. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. Either a positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. Successive calls to `NextVector` (see below) will result in the enumeration of the vectors.

`NextVector(P)`

Given a lattice enumeration process P as created by `ShortVectorsProcess` or `CloseVectorsProcess`, return the next element found in the enumeration.

If the process is for short vectors, the next short vector of the specified region of the lattice is returned together with its norm, or the zero vector together with -1, indicating that the enumeration process has been completed. The vectors are computed up to sign (so that only one of v and $-v$ will be enumerated) and normalized so that the first non-zero entry in each vector is positive.

If the process is for the vectors close to w, the next close vector v whose squared distance $d = (v - w, v - w)$ from w is in the specified range is returned together with

d, or the zero vector together with -1, indicating that the enumeration process has been completed. The close vectors are *not* symmetrical with respect to sign (while short vectors are) so the returned close vectors are not normalized. Note also that to test for completion the second return value should be tested for equality with -1 (or, preferably, the next function `IsEmpty` should be used) since the zero vector could be a valid close vector.

Note that the order of the vectors returned by this function in each case is arbitrary, unlike the previous functions where the resulting sequence or matrix is sorted.

`IsEmpty(P)`

Given a lattice enumeration process P, return whether the process P has found all short or close vectors.

30.8.5 Successive Minima and Theta Series

`SuccessiveMinima(L)`

`SuccessiveMinima(L, k)`

Return the first k successive minima of lattice L, or all of the m successive minima of L if k is omitted, where m is the rank of L. The first k successive minima $M_1, \ldots, M_k$ of a lattice L are defined by the property that $M_1, \ldots, M_k$ are minimal such that there exist linearly independent vectors $l_1, \ldots, l_k$ in L with $(l_i, l_i) = M_i$ for $1 \leq i \leq k$. The function returns a sequence containing the minima M_i and a sequence containing the vectors l_i. The lattice L must be an exact lattice (over $\mathbf{Z}$ or $\mathbf{Q}$). Note that the minima are unique but the vectors are not.

`ThetaSeries(L, n)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Given an integral lattice L and a small positive integer n, return the Theta series $\Theta_L(q)$ of L as a formal power series in q to precision n (i.e., up to and including the coefficient of q^n). The coefficient of q^k in $\Theta_L(q)$ is defined to be the number of vectors of norm k in L. Note that this function needs to enumerate all vectors of L having norm up to and including n, so its application is restricted to lattices where the number of these vectors is reasonably small. The lattice L must be an exact lattice (over $\mathbf{Z}$ or $\mathbf{Q}$). Note that the angle bracket notation should be used to assign a name to the indeterminate for the returned Theta series (e.g., `T<q> := ThetaSeries(L);`).

Example H30E15

We show how a lattice enumeration process can be used to compute the Theta series of a lattice. We write a simple function `Theta` which takes a lattice L and precision n and returns the Theta series of L up to the term q^n just as in the function `ThetaSeries`. The function assumes that the lattice L is integral. We simply loop over the non-zero vectors of norm up to n and count the number of vectors for each norm.

```
> function Theta(L, n)
>     Z := IntegerRing();
>     P := ShortVectorsProcess(L, n);
>     C := [1] cat [0: i in [1 .. n]];
>     while not IsEmpty(P) do
>         v, norm := NextVector(P);
>         C[Z!norm + 1] +:= 2;
>     end while;
>     return PowerSeriesRing(IntegerRing()) ! C;
> end function;
```

We now compute the Theta series up to norm 10 of the Gosset lattice E_8 using the function `Theta`. We compare this Magma-language version with the builtin function `ThetaSeries` (which is much faster of course because of the lack of interpreter overhead etc.).

```
> L := Lattice("E", 8);
> time T<q> := Theta(L, 10);
Time: 0.050
> T;
1 + 240*q^2 + 2160*q^4 + 6720*q^6 + 17520*q^8 + 30240*q^10
> time TT<r> := ThetaSeries(L, 10);
Time: 0.000
> TT;
1 + 240*q^2 + 2160*q^4 + 6720*q^6 + 17520*q^8 + 30240*q^10 + O(q^11)
```

`ThetaSeriesIntegral(L, n)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Restriction of `ThetaSeries` to integral lattices.

30.8.6 Lattice Enumeration Utilities

`SetVerbose("Enum", v)`

(Procedure.) Set the verbose printing level for the lattice enumeration algorithm to be v. Currently the legal values for v are `true`, `false`, 0, and 1 (`false` is the same as 0, and `true` is the same as 1). When the verbose level is non-zero, some information about the current status of the enumeration algorithm is printed out every 15 seconds. This concerns the functions `Minimum`, `CentreDensity`, `CenterDensity`, `Density`, `KissingNumber`, `ShortestVectors`, `ShortestVectorsMatrix`, `ShortVectors`, `ShortVectorsMatrix` and `ThetaSeries`.

`EnumerationCost(L)`

`EnumerationCost(L, u)`

`Prune` SeqEnum *Default :* $[1.0, \ldots, 1.0]$

Estimate the number of nodes in the tree to be visited during the execution of the enumeration algorithm, for the lattice L and an hyperball of squared radius u. If u is not provided, an upper bound to the lattice minimum is used. Since the cost per tree node is relatively constant, this function can be used to obtain a heuristic estimate of the running-time of the enumeration algorithm. Note that in some cases the enumeration may run faster than expected, because the tree to be visited may be shrunk during the execution, as described in [SE94]. Contrary to the enumeration itself, this function runs in time polynomial in the input bit-size (if the value of the `Prune` optional parameter is the default one). It relies on the Gaussian heuristic, which consists in estimating the number of integral points within an n-dimensional body by its volume (see [HS07] for more details).

If the `Prune` parameter is set to $[p_1, \ldots, p_d]$, then `EnumerationCost` estimates the cost of the tree pruned with the strategy described in the introduction of the present section, with pruning coefficients $p_1, \ldots, p_d$. Although the enumeration itself is likely to terminate faster, the estimation of the cost may be significantly more time-consuming.

`EnumerationCostArray(L)`

`EnumerationCostArray(L, u)`

`Prune` SeqEnum *Default :* $[1.0, \ldots, 1.0]$

Estimate the number of nodes in each layer of the tree to be visited during the execution of the enumeration algorithm, for the lattice L and an hyperball of squared radius u. If u is not provided, an upper bound to the lattice minimum is used.

Example H30E16

The algorithm that enumerates short lattice vectors may be very time-consuming. The `EnumerationCost` and `EnumerationCostArray` functions allow one to estimate the running-time on a given enumeration instantiation, before actually running the algorithm. If the running-time estimate is too high, then it is unlikely that the execution will terminate within a reasonable amount of time. A good strategy then consists in reducing the input lattice basis further. If the running-time estimate remains too high, then pruning the enumeration tree should be considered. Both the running-time gain and the "probability" of missing a solution can be estimated.

```
> B:=RMatrixSpace(IntegerRing(), 50, 51) ! 0;
> for i := 1 to 50 do B[i][1] := RandomBits(1000); end for;
> for i := 1 to 50 do B[i][i+1] := 1; end for;
> B := LLL(B);
> EnumerationCost (Lattice(B));
4.06E18
```

The value above is an estimate of the number of nodes in the enumeration tree that corresponds to the computation of the shortest non-zero vectors in the lattice spanned by the rows of B. This is much too high to have a chance to terminate. Reducing B further allows us to compute these shortest non-zero vectors.

```
> B:=LLL(B:Delta:=0.999);
> EnumerationCost(Lattice(B));
1.03E13
> B:=LLL(B:Delta:=0.999, DeepInsertions);
> EnumerationCost(Lattice(B));
7.43E7
> time _:=ShortestVectors(Lattice(B));
3.660
```

In larger dimensions, reducing the lattice further may not prove sufficient to make the enumeration reasonably tractable. Then pruning the enumeration tree can be considered.

```
>  B:=RMatrixSpace(IntegerRing(), 65, 66) ! 0;
>  for i := 1 to 65 do B[i][1] := RandomBits(1000); end for;
>  for i := 1 to 65 do B[i][i+1] := 1; end for;
>  B := LLL(B:Delta:=0.999);
>  B := LLL(B:Delta:=0.999, DeepInsertions);
>  EnumerationCost(Lattice(B));
2.77E12
> p:=[1.0: i in [1..65]];
> for i:=10 to 55 do p[i] := (100-i)/90.; end for;
> for i:=56 to 65 do p[i]:=0.5; end for;
> EnumerationCost(Lattice(B):Prune:=p);
1.75E10
> time _:=ShortestVectors(Lattice(B):Prune:=p);
Time: 422.120
```

Assuming that the number of visited tree nodes per time unit remains roughly constant, the execution would have taken around 18 hours without pruning. It is possible to estimate the likeliness of not missing the optimal solution, by looking at the first coefficient returned by `EnumerationCostArray`. Assuming the Gaussian heuristic, this is the ratio between the first coefficients of `EnumerationCostArray` with and without the pruning table.

```
>  t1:= EnumerationCostArray(Lattice(B):Prune:=p)[1];
>  t2:= EnumerationCostArray(Lattice(B))[1];
>  t1/t2;
0.992
```

Experimentally, it seems that pruning coefficients that decrease linearly (except for the first and last indices) provide good trade-offs between efficiency gain and success likeliness.

30.9 Theta Series as Modular Forms

The theta series of an integral lattice L is (the q-expansion of) a modular form whose weight is half the dimension of the lattice, and whose level and nebentypus are determined by the quotient $L^{\#}/L$. The space of forms with given weight, level and character is finite dimensional, which means the theta series is uniquely characterised as an element of that space from knowledge of finitely many of its coefficients.

The routine described below carries this out explicitly: given an integral lattice, it returns an element of a Magma space of modular forms, Moreover this is done with the least possible effort spent determining coefficients via lattice enumeration. Several ideas are used here, the most important of which is that linear constraints can be obtained from knowing coefficients of the theta series of L *and* of its partial dual lattices; in practice, one needs a certain number of coefficients in total, and the `EnumerationCost` functionality is useful for balancing how many to compute for each of the duals.

We say L is *q-modular* if it is isomorphic to its qth partial dual L_q. Knowledge that modularities exist is of some use in the algorithm, because q-modularity clearly implies that L and L_q have the same theta series.

Normalisation of theta series: We use a normalisation that is most natural in this context, in order that the theta series is a modular form on $\Gamma_0(N)$ with the level N as small as possible. In this section, the theta series of an even integral lattice L will mean $\Sigma_{v\in L}\, q^{1/2|v|^2}$, while the theta series of an odd integral lattice L will mean $\Sigma_{v\in L}\, q^{|v|^2}$, where $q = e^{2i\pi z}$.

```
ThetaSeriesModularFormSpace(L)
```

Given an integral lattice L, this returns the space of modular forms which contains the theta series of L. Note: the theta series is normalised as described above.

```
ThetaSeriesModularForm(L)
```

`KnownTheta`	RngSerPowElt	*Default :*
`KnownDualThetas`	SeqEnum[Tup]	*Default :*
`KnownModularities`	Set[RngIntElt]	*Default :*
`ComputeModularities`	Set[RngIntElt]	*Default :*

Given an integral lattice L, this returns the theta series of L as a modular form in the appropriate space: more precisely, in `ThetaSeriesModularFormSpace(L)`. Note: the theta series is normalised as described above.

If some coefficients of the theta series of L are already known, this information may be specified by setting the optional argument `KnownTheta` to be a power series f, indicating that L has theta series equal to f up to the precision of f.

More generally, if coefficients are known for any partial dual lattices, these may be specified by setting `KnownDualThetas`. This argument must be a sequence of tuples of the form $< q, f_q >$, indicating that the qth partial dual L_q has theta series equal to f_q up to the precision of f_q.

If L is known to possess modularities, the optional argument `KnownModularities` may be set equal to any set of integers q such that L is q-modular (as defined above).

In addition, `ComputeModularities` may be specified to control whether the function checks for q-modularities (for possible q, that are not listed as `KnownModularities`); its value may be a boolean, or a set of integers.

30.10 Voronoi Cells, Holes and Covering Radius

The functions in this section compute the Voronoi cell of a lattice around the origin and associated information. Note that the computation of the Voronoi cell is of truly exponential complexity and is therefore the use of these functions is restricted to small dimensions (up to about 10). See [JC98] for the relevant definitions.

A lattice to which any of these functions are applied must be an exact lattice (over **Z** or **Q**).

`VoronoiCell(L)`

Return the Voronoi cell of the lattice L around the origin which is the convex polytope consisting of all the points closer to the origin than to any other lattice point of L. This function returns three values:

(a) A sequence V of vectors which are the vertices of the Voronoi cell.

(b) A set E of pairs, where each pair $\{i, j\}$ represents an edge connecting $V[i]$ and $V[j]$.

(c) A sequence P of vectors defining the relevant hyperplanes. A vector p corresponds to the hyperplane given by $(x, p) = \mathrm{Norm}(p)/2$.

`VoronoiGraph(L)`

Return a graph having the vertices and edges of the Voronoi cell of the lattice L as vertices and edges, respectively.

`Holes(L)`

Return a sequence of vectors which are the holes of the lattice L. The holes are defined to be the vertices of the Voronoi cell around the origin. Note that this involves computing the Voronoi cell of L around the origin.

`DeepHoles(L)`

Return a sequence of vectors which are the deep holes of the lattice L. The deep holes are defined to be the holes of maximum norm and are points of maximum distance to all lattice points. Note that this involves computing the Voronoi cell of L around the origin.

`CoveringRadius(L)`

Return the squared covering radius of the lattice L, which is the norm of the deep holes of L. Note that this involves computing the Voronoi cell of L around the origin.

`VoronoiRelevantVectors(L)`

Return the Voronoi relevant hyperplanes (as a set of vectors) of the Voronoi cell of L around the origin. Note that this is the same as the third return value of the `VoronoiCell` intrinsic. However, it is usually much faster since it does not compute the Voronoi cell of L. The algorithm employed is [AEVZ02, Section C].

Example H30E17

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: 1.740
> #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.

30.11 Orthogonalization

The functions in this section perform orthogonalization and orthonormalization of lattice bases over the field of fractions of the base ring. Note that this yields a basis orthogonalization of the space in which a lattice embeds; in contrast `OrthogonalDecomposition` returns a decomposition into orthogonal components over the base ring. Basis orthogonalization is equivalent to diagonalization of the inner product matrix of a space.

`Orthogonalize(M)`

Given a basis matrix M over a subring R of the real field, compute a matrix N which is row-equivalent over to M over the field of fractions K of R, but whose rows are orthogonal (i.e., NN^{tr} is a diagonal matrix). This function returns three values:

(a) An orthogonalized matrix N in row-equivalent to X over K;

(b) An invertible matrix T in the matrix ring over K whose degree is the number of rows of M such that $TM = N$;

(c) The rank of M.

`Diagonalization(F)`

`OrthogonalizeGram(F)`

Given a symmetric n×n matrix F over R, where R is a subring of the real field, compute a *diagonal* matrix G such that $G = TFT^{tr}$ for some invertible matrix T over K, where K is the field of fractions of R. F need not have rank n. This function returns three values:

(a) A diagonal matrix G defined over R;

(b) An invertible n×n matrix T over K such that $G = TFT^{tr}$;

(c) The rank of F.

`Orthogonalize(L)`

For a lattice L, return a new lattice having the same Gram matrix as L but embedded in an ambient space with diagonal inner product matrix.

`Orthonormalize(M, K)`

`Cholesky(M, K)`

`Orthonormalize(M)`

`Cholesky(M)`

For a symmetric, positive definite matrix M, and a real field K, return a lower triangular matrix T over K such that $M = TT^{tr}$. The algorithm must take square roots so the result is returned as a matrix over the real field K. If the real field K is omitted, K is taken to be the default real field. Note that this function takes a Gram matrix M, *not* a basis matrix as in the previous functions.

`Orthonormalize(L, K)`

`Cholesky(L, K)`

`Orthonormalize(L)`

`Cholesky(L)`

Given a lattice L with Gram matrix F, together with a real field K, return a new lattice over K which has the same Gram matrix F as L but has the standard Euclidean inner product. (This will involve taking square roots so that is why the result must be over a real field.) The argument for the real field K may be omitted, in which case K is taken to be the current default real field. This function is equivalent to the invocation `LatticeWithBasis(Orthonormalize(GramMatrix(L), K))`. It is sometimes more convenient to work with the resulting lattice since it has the standard Euclidean inner product.

Example H30E18

As an example for a lattice with non-trivial basis and inner product matrices we choose the dual lattice of the 12-dimensional Coxeter-Todd lattice. We compute the inner products of all pairs of shortest vectors and notice that this gets faster after changing to an isomorphic lattice with weighted standard Euclidean inner product.

```
> L := Dual(CoordinateLattice(Lattice("Kappa", 12)));
> SL := ShortestVectors(L);
> SL := SL cat [ -v : v in SL ]; #SL;
756
> time { (v,w) : v,w in SL };
{ -4, -2, -1, 0, 1, 2, 4 }
Time: 7.120
> M := Orthogonalize(L);
> SM := ShortestVectors(M);
> SM := SM cat [ -v : v in SM ]; #SM;
756
> time { (v,w) : v,w in SM };
{ -4, -2, -1, 0, 1, 2, 4 }
```

```
Time: 1.300
```

30.12 Testing Matrices for Definiteness

The functions in this section test matrices for positive definiteness, etc. They may applied to any symmetric matrix over a real subring (i.e., $\mathbf{Z}$, $\mathbf{Q}$, or a real field). Each function works by calling the function `OrthogonalizeGram` on its argument and then determining whether the resulting diagonal matrix has the appropriate form.

`IsPositiveDefinite(F)`

Given a symmetric matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, return whether F is positive definite, i.e., whether $vFv^{tr} > 0$ for all non-zero vectors $v \in \mathbf{R}^n$.

`IsPositiveSemiDefinite(F)`

Given a symmetric matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, return whether F is positive semi-definite, i.e., whether $vFv^{tr} \geq 0$ for all non-zero vectors $v \in \mathbf{R}^n$.

`IsNegativeDefinite(F)`

Given a symmetric matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, return whether F is negative definite, i.e., whether $vFv^{tr} < 0$ for all non-zero vectors $v \in \mathbf{R}^n$.

`IsNegativeSemiDefinite(F)`

Given a symmetric matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, return whether F is negative semi-definite, i.e., whether $vFv^{tr} \leq 0$ for all non-zero vectors $v \in \mathbf{R}^n$.

30.13 Genera and Spinor Genera

The genus of an exact lattice has a distinct type `SymGen` which holds a representative lattice, and the local data defining the genus. Each genus consists of 2^n spinor genera, for some integer n, typically 1. The spinor genera share the same type `SymGen`. Unlike the genus, the spinor genus is not determined solely by the local data of the genus, so the cached representative is necessary to define the spinor class.

Equality testing of genera is fast, since this requires only a comparison of the canonical local information. It is also possible to enumerate representatives of all equivalences classes in a genus or spinor genus. This is done by a process of exploration of the p-neighbour graph, for an appropriate prime p. The neighbouring functions can be applied to individual lattices to find p-neighbours or the closure under the p-neighbour process. Functions for computing and comparing the local p-adic equivalence classes of lattices, mediated by the type `SymGenLoc`.

30.13.1 Genus Constructions

`Genus(L)`

`Genus(G)`

Given an exact lattice L or a spinor genus G this function returns the genus of L. If given a genus the function returns G itself.

`SpinorGenus(L)`

Given an exact lattice L, returns the spinor genus of L.

`SpinorGenera(G)`

Given a genus G, returns the sequence of spinor genera. If G is a spinor genus, then this function returns the sequence consisting of G itself.

30.13.2 Invariants of Genera and Spinor Genera

`Representative(G)`

Returns a representative lattice for the genus symbol G.

`IsSpinorGenus(G)`

Returns `true` if and only if G is a spinor genus. This is the negation of `IsGenus(G)`.

`IsGenus(G)`

Returns `true` if and only if G is a genus. This is the negation of `IsSpinorGenus(G)`.

`Determinant(G)`

Returns the determinant of the genus symbol G.

`LocalGenera(G)`

Returns the sequence of p-adic genera of the genus symbol G.

`Representative(G)`

Returns a representative lattice for the genus symbol G.

`G1 eq G2`

Given two genus symbols, return `true` if and only if they represent the same genus. This computation is fast for genera, but currently for spinor genera invokes a call to `Representatives`.

`#G`

The number of isometry classes in the genus or spinor genus G.

Enumeration of isometry classes is done by an explicit call to `Representatives`, so that `#G` is an expensive computation.

`SpinorCharacters(G)`

Return the spinor characters of the genus symbol G as a sequence of Dirichlet characters whose kernels intersect exactly in the group of automorphous numbers. Consult Conway and Sloane [JC98] for precise definitions and significance of the spinor kernel and automorphous numbers.

`SpinorGenerators(G)`

Return the spinor generators of the genus symbol G as a sequence of primes which generate the group of spinor norms. The primes generate a group dual to that generated by the spinor characters.

`AutomorphousClasses(L,p)`

`AutomorphousClasses(G,p)`

A set of integer representatives of the p-adic square classes in the image of the spinor norm of the lattice L (respectively the genus symbol G).

`IsSpinorNorm(G,p)`

Returns `true` if and only if p is coprime to 2 and the determinant, and p is the norm of an element of the spinor kernel of G.

30.13.3 Invariants of p-adic Genera

`Prime(G)`

Return the prime p for which G represents the p-adic genus.

`Representative(G)`

Returns a canonical representative lattice of the p-adic genus G, with Gram matrix in Jordan form. For odd p the Jordan form is diagonalized.

`Determinant(G)`

This function returns a canonical p-adic representative of the determinant of the p-adic genus G. The determinant is well-defined only up to squares.

`Dimension(G)`

Return the dimension of the p-adic genus G.

`G1 eq G2`

Given local genus symbols $G1$ and $G2$, return `true` if and only if they have the same prime and the same canonical Jordan form.

30.13.4 Neighbour Relations and Graphs

`Neighbour(L, v, p)`

`Neighbor(L, v, p)`

Let L be an integral lattice, p a prime which does not divide Determinant(L) and v a vector in $L \setminus pL$ with $(v, v) \in p^2\mathbf{Z}$. The p-neighbour of L with respect to v is the lattice generated by L_v and $p^{-1}v$, where $L_v := \{x \in L | (x, v) \in p\mathbf{Z}\}$.

See [Kne57] for the original definition and [SP91] for a generalization of the neighbouring method.

`Neighbours(L, p)`

`Neighbors(L, p)`

For an integral lattice L and prime p, returns the sequence of p-neighbours of L.

`NeighbourClosure(L, p)`

`NeighborClosure(L, p)`

`Bound` RngIntElt *Default :* 2^{32}

For an integral lattice L and prime p, returns the sequence of lattices obtained by transitive closure of the p-neighbours of L.

Note that neighbours with respect to two vectors v_1, v_2 whose images in L/pL lie in the same projective orbit of Aut(L) on L/pL are isometric. Therefore only projective orbit representatives of the action of Aut(L) on L/pL are used. The large number of orbits restricts the complexity of this algorithm, hence the function gives an error if $p^{\mathrm{Rank}(L)}$ is greater than `Bound`, by default set to 2^{32}.

```
GenusRepresentatives(L)
SpinorRepresentatives(L)
Representatives(G)
```

`Bound`	RngIntElt	*Default :* 2^{32}
`Depth`	RngIntElt	*Default :*

For an exact lattice L with genus or spinor genus G, this function enumerates the isometry classes in G by constructing the p-neighbour closure (up to isometry). This construction used using an appropriate prime or primes p not dividing the determinant of L. For the genus, sufficiently many primes p are chosen to generate the full image, modulo the spinor kernel, of each character defining the spinor kernel. The parameters are exactly as for the `NeighbourClosure` function.

```
AdjacencyMatrix(G,p)
```

For a genus or spinor genus G, this function determines the adjacency matrix of the p-neighbour graph on the representative classes for G. The integer p must be prime, and if G is a spinor genus, then an error ensues if p is not an automorphous number for G.

Example H30E19

We construct the root lattice E_8 (the unique even unimodular lattice of dimension 8) as a 2-neighbour of the 8-dimensional standard lattice.

```
> Z8 := StandardLattice(8);
> v := Z8 ! [1,1,1,1,1,1,1,1];
> E8 := Neighbour(Z8, v, 2);
> E8;
Lattice of rank 8 and degree 8
Basis:
( 2  0  0  0  0  0  0  2)
( 2  0  0  0  0  0  0 -2)
( 1  1 -1  1  1 -1  1  1)
( 1  1 -1 -1 -1 -1 -1 -1)
( 1 -1 -1 -1 -1 -1  1 -1)
( 0  0  2  0  0  0  0  2)
( 0  0  0  0  2  0  0  2)
( 0  0  0  0  0  2  0  2)
Basis denominator: 2
```

The so-obtained lattice is in fact identical to the one returned by the standard construction.

```
> L := Lattice("E", 8);
> L;
Lattice of rank 8 and degree 8
Basis:
( 4  0  0  0  0  0  0  0)
```

```
(-2  2  0  0  0  0  0  0)
( 0 -2  2  0  0  0  0  0)
( 0  0 -2  2  0  0  0  0)
( 0  0  0 -2  2  0  0  0)
( 0  0  0  0 -2  2  0  0)
( 0  0  0  0  0 -2  2  0)
( 1  1  1  1  1  1  1  1)
Basis Denominator: 2
> E8 eq L;
true
```

Example H30E20

In this example we enumerate representatives for the genus of the Coxeter-Todd lattice, performing the major steps manually. The whole computation can be done simply by calling the `GenusRepresentatives` function but the example illustrates how the function actually works.

We use a combination of the automorphism group, isometry and neighbouring functions. The idea is that the neighbouring graph spans the full genus which therefore can be computed by successively generating neighbours and checking them for isometry with already known ones. The automorphism group comes into play, since neighbours with respect to vectors in the same projective orbit under the automorphism group are isometric.

```
> L := CoordinateLattice(Lattice("Kappa", 12));
> G := AutomorphismGroup(L);
> G2 := ChangeRing(G, GF(2));
> O := LineOrbits(G2);
> [ Norm(L!Rep(o).1) : o in O ];
[ 4, 8, 10 ]
```

Hence only the first and second orbits give rise to a 2-neighbour. To obtain an even neighbour, the second vector has to be adjusted by an element of $2 * L$ such that it has norm divisible by 8.

```
> v1 := L ! Rep(O[1]).1;
> v1 +:= 2 * Rep({ u : u in Basis(L) | (v1,u) mod 2 eq 1 });
> v2 := L ! Rep(O[2]).1;
> Norm(v1), Norm(v2);
16 8
> L1 := Neighbour(L, v1, 2);
> L2 := Neighbour(L, v2, 2);
> bool := IsIsometric(L, L1); bool;
true
> bool := IsIsometric(L, L2); bool;
false
```

So we obtain only one non-isometric even neighbour of L. To obtain the full genus we can now proceed with $L2$ in the same way, and do this with the following function `EvenGenus`. Note that this function is simply one component of the function `GenusRepresentatives`.

```
> function EvenGenus(L)
>      // Start with the lattice L
>      Lambda := [ CoordinateLattice(LLL(L)) ];
>      cand := 1;
>      while cand le #Lambda do
>          L := Lambda[cand];
>          G := ChangeRing( AutomorphismGroup(L), GF(2) );
>          // Get the projective orbits on L/2L
>          O := LineOrbits(G);
>          for o in O do
>              v := L ! Rep(o).1;
>              if Norm(v) mod 4 eq 0 then
>                  // Adjust the vector such that its norm is divisible by 8
>                  if not Norm(v) mod 8 eq 0 then
>                      v +:= 2 * Rep({ u : u in Basis(L) | (v,u) mod 2 eq 1 });
>                  end if;
>                  N := LLL(Neighbour(L, v, 2));
>                  new := true;
>                  for i in [1..#Lambda] do
>                      if IsIsometric(Lambda[i], N) then
>                          new := false;
>                          break i;
>                      end if;
>                  end for;
>                  if new then
>                      Append(~Lambda, CoordinateLattice(N));
>                  end if;
>              end if;
>          end for;
>          cand +:= 1;
>      end while;
>      return Lambda;
> end function;
>
> time Lambda := EvenGenus(L);
Time: 9.300
> #Lambda;
10
> [ Minimum(L) : L in Lambda ];
[ 4, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
> &+[ 1/#AutomorphismGroup(L) : L in Lambda ];
4649359/4213820620800
```

We see that the genus consists of 10 classes of lattices where only the Coxeter-Todd lattice has minimum 4 and get the mass of the genus as 4649359/4213820620800.

30.14 Attributes of Lattices

This section lists various attributes of lattices which can be examined and set by the user. This allows low-level control of information stored in lattices. Note that when an attribute is set, only minimal testing can be done on the value so if an incorrect value is set, unpredictable results may occur. Note also that if an attribute is not set, referring to it in an expression (using the ‘ operator) will *not* trigger the calculation of it (while intrinsic functions do); rather an error will ensue. Use the `assigned` operator to test whether an attribute is set.

`L‘Minimum`

The attribute for the minimum of a rational or integer lattice L. If the attribute `L‘Minimum` is examined, either the minimum is known so it is returned or an error results. If the attribute `L‘Minimum` is set by assignment, it must be a positive number equal to the minimum of the lattice. Magma will not check that this is correct since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value.

`L‘MinimumBound`

The attribute for an upper bound to the minimum of a rational or integer lattice L. If the attribute `L‘MinimumBound` is examined, either an upper bound to the minimum is known so it is returned or an error results. If the attribute `L‘MinimumBound` is set by assignment, it must be a positive number greater than or equal to the minimum of the lattice. Magma will not check that this is correct since that may be very time-consuming. If the attribute is already set, the new value must be the same as or smaller than the old value.

30.15 Database of Lattices

Magma includes a database containing most of the lattices explicitly presented in the Catalogue of Lattices maintained by Neil J.A. Sloane and Gabriele Nebe [NS01b].

Many standard lattices included in the Sloane & Nebe catalogue are not in the database as they may be obtained by applying Magma’s standard lattice creation functions. Also omitted from the database are a small number of catalogued lattices defined over rings other than **Z** or **Q**.

The information available for any given lattice in the catalogue varies considerably. A similar variety is found in the Magma database version, although some data (generally either easily computable or rarely available in the catalogue) is omitted.

Where the Magma database does retain data, it is not altered from the data in the catalogue. Thus the caveat which comes with that catalogue remains relevant: “Warning! Not all the entries have been checked!”

A second version of the Lattice Database has been made available with Version 2.16 of Magma. It adds a few more lattices, contains more information about automorphism groups, and adds Θ-series as attributes. Furthermore, it removes some duplicates. The user should be warned that the numbering of lattices (and naming in some cases) differs between the two versions. The newer version, however, does not contain information about any Hermitian structure at the current time.

The entries of the database can be accessed in three ways:

(i) the i-th entry of the database can be requested;

(ii) the i-th entry of a particular dimension d can be specified;

(iii) the desired entry can be denoted by its name N. This name is specified exactly as in the catalogue, including all punctuation and whitespace. In the rare event that two or more entries share a single name, particular entries may be distinguished by supplying an integer i in addition to N, to denote the i-th entry with name N.

30.15.1 Creating the Database

`LatticeDatabase()`

This function returns a database object which contains information about the database.

30.15.2 Database Information

This section gives the functions that enable the user to find out what is in the database.

`#D`

`NumberOfLattices(D)`

Returns the number of lattices stored in the database.

`LargestDimension(D)`

Returns the largest dimension of any lattice in the database.

`NumberOfLattices(D, d)`

Returns the number of lattices of dimension d stored in the database.

`NumberOfLattices(D, N)`

Returns the number of lattices named N stored in the database.

`LatticeName(D, i)`

Return the name and dimension of the i-th entry of the database D.

`LatticeName(D, d, i)`

Return the name and dimension of the i-th entry of dimension d of the database D.

`LatticeName(D, N)`

Return the name and dimension of the first entry of the database with name N.

`LatticeName(D, N, i)`

Return the name and dimension of the i-th entry of the database with name N.

Example H30E21

We find out the names of the database entries.

```
> D := LatticeDatabase();
> NumberOfLattices(D);
700
```

The database contains 700 lattices. We get the set of all names in the database.

```
> names := {LatticeName(D,i): i in [1..#D]};
> #names;
673;
> Random(names);
S4(5):2
> NumberOfLattices(D, "S4(5):2");
1
```

There are 673 names, so 27 repeated names in the database, but only one with name "S4(5):2".

30.15.3 Accessing the Database

The following functions retrieve lattice information from the database.

`Lattice(D, i: parameters)`

`Lattice(D, d, i: parameters)`

`Lattice(D, N: parameters)`

`Lattice(D, N, i: parameters)`

`TrustAutomorphismGroup` BOOL *Default :* `true`

Returns the i-th entry (of dimension d or name N) from the database D as a lattice L.

If the `TrustAutomorphismGroup` parameter is assigned `false`, then any data which claims to be the automorphism group will not be stored in L.

`LatticeData(D, i)`

`LatticeData(D, d, i)`

`LatticeData(D, N)`

`LatticeData(D, N, i)`

Returns a record which contains all the information about the i-th lattice stored in the database D (of dimension d or name N). The automorphism group is returned separately from the lattice and not stored in it.

Example H30E22

We look up a lattice in the database. There are 19 lattices of dimension 6 in the database. We get the 10th.

```
> D := LatticeDatabase();
> NumberOfLattices(D, 6);
19
> L := Lattice(D, 6, 10);
> L;
Standard Lattice of rank 6 and degree 6
Minimum: 4
Inner Product Matrix:
[4 1 2 2 2 2]
[1 4 2 2 2 2]
[2 2 4 1 2 2]
[2 2 1 4 2 2]
[2 2 2 2 4 1]
[2 2 2 2 1 4]
```

There may be more information stored than just what is returned by the `Lattice` function. We get the record containing all the stored lattice data.

```
> R := LatticeData(D, 6, 10);
> Format(R);
recformat<name, dim, lattice, minimum, kissing_number,
is_integral, is_even, is_unimodular, is_unimodular_hermitian,
modularity, group_names, group, group_order,
hermitian_group_names, hermitian_group, hermitian_group_order,
hermitian_structure>
```

This lists all possible fields in the record. They may or may not be assigned for any particular lattice.

```
> R`lattice eq L;
true
> R`name;
A6,1
> assigned R`kissing_number;
true
> R`kissing_number;
42
> assigned R`group;
false
> A := AutomorphismGroup(L);
> A : Minimal;
MatrixGroup(6, Integer Ring) of order 96 = 2^5 * 3
```

The result of the `Lattice` call is equal to the `lattice` field of the data record. The kissing number was stored, but the automorphism group wasn't. We computed the group (as a matrix group over the integers) and found it has order 96.

30.15.4 Hermitian Lattices

There are a few facilities for computing with Hermitian lattices over an imaginary quadratic field or a quaternion algebra. However, these functions apply to a Gram matrix, and not a lattice *per se.* The main application is for automorphism groups that preserve a structure.

`HermitianTranspose(M)`

Given a matrix over an imaginary quadratic field or a quaternion algebra, return the conjugate transpose.

`ExpandBasis(M)`

Given a matrix over an imaginary quadratic field or a quaternion algebra, expand it to a basis over the rationals.

`HermitianAutomorphismGroup(M)`
`QuaternionicAutomorphismGroup(M)`

Given a conjugate symmetric Gram matrix, compute the automorphism group.

Various functions for matrix groups over associative algebras are available here, such as `CharacterTable` and `IsConjugate` which simply use a re-writing over the rationals, and `InvariantForms` which after using `GHom` needs to restrict to elements fixed by the quaterionic structure. Finally, there is `QuaternionicGModule` which will split a G-module over a quaternionic structure.

`InvariantForms(G)`

Given a matrix group over an associative algebra or an imaginary quadratic field, return a basis for the forms fixed by it.

`QuaternionicGModule(M, I, J)`

Given a G-module M and I and J in the endomorphism algebra that anti-commute and whose squares are scalars, write G over the quaternionic structure given by I and J.

`MooreDeterminant(M)`

Given a conjugate-symmetric matrix over a quaternion algebra, compute the Moore determinant. This is the "normal" determinant, which is well-defined here because all the diagonal elements are rational, and thus there is no ambiguity between left/right division.

Example H30E23

We construct the Coxeter-Todd lattice over $Q_{3,\infty}$ starting with the group $SU(3,3)$.

```
> G := SU(3, 3);
> chi := CharacterTable(G)[2];
> M := GModule(chi,Integers());
> E := EndomorphismAlgebra (M);
> while true do r:=&+[Random([-2..2])*E.i : i in [1..4]];
>                if r^2 eq -1 then break; end if; end while;
> while true do s:=&+[Random([-2..2])*E.i : i in [1..4]];
>                if s^2 eq -3 and r*s eq -s*r then break; end if; end while;
> MM := QuaternionicGModule(M, r, s);
> Discriminant(BaseRing(MM));
3
> MG := MatrixGroup(MM);
> IF := InvariantForms(MG); IF;
[
    [1 -1/2*i + 1/6*k 1/3*k]
    [1/2*i - 1/6*k 1 -1/3*j]
    [-1/3*k 1/3*j 1]
]
> assert IsIsomorphic(G, MG);
```

Example H30E24

We compute the quaternionic automorphism group for the Leech lattice.

```
> A<i,j,k> := QuaternionAlgebra<Rationals()|-1,-1>;
> v := [];
> v[1] := [2+2*i,0,0,0,0,0]; /* from Wilson's paper */
> v[2] := [2,2,0,0,0,0];
> v[3] := [0,2,2,0,0,0];
> v[4] := [i+j+k,1,1,1,1,1];
> v[5] := [0,0,1+k,1+j,1+j,1+k];
> v[6] := [0,1+j,1+j,1+k,0,1+k];
> V := [Vector(x) : x in v];
> W := [Vector([Conjugate(x) : x in Eltseq(v)]): v in V];
> M6 := Matrix(6,6,[(V[i],W[j])/2 : i,j in [1..6]]); /* 6-dim over A */
> time Q := QuaternionicAutomorphismGroup(M6);
> assert #Q eq 503193600;
```

The same can be done for the Coxeter-Todd lattice.

```
> A<i,j,k> := QuaternionAlgebra<Rationals()|-1,-3>;
> a := (1+i+j+k)/2;
> M3 := Matrix(3,3,[2,a,-1, Conjugate(a),2,a, -1,Conjugate(a),2]);
> time Q := QuaternionicAutomorphismGroup(M3);
> assert #Q eq 12096;
```

One can also compute the automorphism group over the Eisenstein field, using `InvariantForms` on the realisation of this group as `ShephardTodd(34)`.

```
> G := ShephardTodd(34);
> IF := InvariantForms(G); // scaled Coxeter-Todd over Q(sqrt(-3))
> A := HermitianAutomorphismGroup(IF[1]);
> assert IsIsomorphic(A,G);
```

30.16 Bibliography

[**AEVZ02**] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.

[**Ajt98**] Miklós Ajtai. The Shortest Vector Problem in L2 is NP-hard for Randomized Reductions (Extended Abstract). In *Proceedings of the 30th Symposium on the Theory of Computing (STOC 1998)*, pages 10–19. ACM, 1998.

[**Akh02**] Ali Akhavi. Random lattices, threshold phenomena and efficient reduction algorithms. *Theoretical Computer Science*, 287(2):359–385, 2002.

[**dW87**] Benne M.M. de Weger. Solving exponential Diophantine equations using lattice basis reduction algorithms. *J. Number Th.*, 26:325–367, 1987.

[**FP83**] U. Fincke and M. Pohst. A procedure for determining algebraic integers of given norm. In *EUROCAL*, volume 162 of *LNCS*, pages 194–202. Springer, 1983.

[**HPP06**] F. Hess, S. Pauli, and M. Pohst, editors. *ANTS VII*, volume 4076 of *LNCS*. Springer-Verlag, 2006.

[**HS07**] Guillaume Hanrot and Damien Stehlé. Improved Analysis of Kannan's Shortest Lattice Vector Algorithm (Extended Abstract). In *Advances in cryptology—CRYPTO 2007*, volume 4622 of *LNCS*, pages 170–186. Springer, 2007.

[**JC98**] N.J.A. Sloane J.H. Conway. *Sphere Packings, Lattices and Groups*, volume 290 of *Grundlehren der Mathematischen Wissenschaften*. Springer, New York–Berlin–Heidelberg, 3rd edition, 1998.

[**Kan83**] R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the 15th Symposium on the Theory of Computing (STOC 1983)*, pages 99–108. ACM, 1983.

[**Kne57**] M. Kneser. Klassenzahlen indefiniter quadratischer Formen. *Archiv Math.*, 8:241–250, 1957.

[**LLL82**] Arjen K. Lenstra, Hendrik W. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[**MG02**] Daniele Micciancio and Shafi Goldwasser. *Complexity of lattice problems: a cryptographic perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 2002.

[**NS01a**] G. Nebe and N.J.A. Sloane. The Catalogue of Lattices. URL:http://www.research.att.com/∼njas/lattices/, 2001.

[**NS01b**] Gabriele Nebe and Neil J.A. Sloane. A Catalogue of Lattices. URL:http://akpublic.research.att.com/∼njas/lattices/index.html, 2001.

[**NS06**] Phong Nguyen and Damien Stehlé. LLL on the Average. In Hess et al. [HPP06], pages 238–256.

[**NS09**] Phong Nguyen and Damien Stehlé. An LLL Algorithm with Quadratic Complexity. *SIAM Journal on Computing*, 39(3):874–903, 2009.

[**Poh87**] Michael Pohst. A Modification of the LLL Reduction Algorithm. *J. Symbolic Comp.*, 4(1):123–127, 1987.

[**Pro**] The SPACES Project. MPFR, a LGPL-library for multiple-precision floating-point computations with exact rounding. URL:http://www.mpfr.org/.

[**PS08**] Xavier Pujol and Damien Stehlé. Rigorous and efficient short lattice vectors enumeration. In *Advances in Cryptology—AsiaCrypt 2008*, LNCS. Springer, 2008.

[**SE94**] Claus-Peter Schnorr and Michael Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Mathematics of Programming*, 66:181–199, 1994.

[**SH95**] Claus-Peter Schnorr and Horst Helmut Hörner. Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction. In *Advances in Cryptology—EuroCrypt 1995*, volume 921 of *LNCS*, pages 1–12. Springer-Verlag, 1995.

[**Sho**] Victor Shoup. NTL, Number Theory C++ Library. URL:http://www.shoup.net/ntl/.

[**Sim05**] Denis Simon. Solving quadratic equations using reduced unimodular quadratic forms. *Math. Comp.*, 74(251):1531–1543 (electronic), 2005.

[**SP91**] Rainer Schulze-Pillot. An algorithm for computing genera of ternary and quaternary quadratic forms. In Stephen M. Watt, editor, *Proceedings ISSAC'91*, pages 134–143, Bonn, 1991.

[**Ste09**] Damien Stehlé. *Floating-point LLL: theoretical and practical aspects.* Springer-Verlag, 2009. To appear.

[**vEB81**] Peter van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical report 81-04, Mathematisch Instituut, Universiteit van Amsterdam, 1981.

31 LATTICES WITH GROUP ACTION

Chapter 31
LATTICES WITH GROUP ACTION

31.1 Introduction

In Magma, a G-lattice L is a lattice upon which a finite integral matrix group G acts by right multiplication. Magma allows various computations with lattices associated with finite integral matrix groups by use of G-lattices.

The computation of the automorphism group of a lattice (i.e. the largest matrix group that acts on the lattice) and the testing of lattices for isometry is performed within Magma by a search designed by Bill Unger, which is based on the Plesken-Souvignier backtrack algorithm [PS97], together with ordered partition methods. Optionally, this may be combined with orthogonal decomposition code of Gabi Nebe.

If G is a finite integral matrix group, then Magma uses Plesken's centering algorithm ([Ple74]) to construct all G-invariant sublattices of a given G-lattice L. The lattice of G-invariant sublattices of L can be explored much like the lattice of submodules over finite fields.

31.2 Automorphism Group and Isometry Testing

The functions in this section compute the automorphism group of a lattice and test lattices for isometry. Currently the lattices to which these functions are applied must be exact (over **Z** or **Q**).

```
AutomorphismGroup(L)
```

`Stabilizer`	RngIntElt	*Default :* 0
`BacherDepth`	RngIntElt	*Default :* -1
`Generators`	[GrpMatElt]	*Default :*
`NaturalAction`	Bool	*Default :* `false`
`Decomposition`	Bool	*Default :* `false`
`Vectors`	Mtrx	*Default :*

This function computes the automorphism group G of a lattice L which is defined to be the group of those automorphisms of the **Z**-module underlying L which preserve the inner product of L. L must be an exact lattice (over **Z** or **Q**). The group G is returned as an integral matrix group of degree m acting on the *coordinate* vectors of L by multiplication where m is the rank of L. The coordinate vectors of L are the elements of the coordinate lattice C of L which has the same Gram matrix as L, but standard basis of degree m (C can be created using the function `CoordinateLattice`). G does not act on the elements of L, since there is no natural matrix action of the automorphism group on L in the case that the rank of L is less than its degree. However, if the rank of L equals its degree, then the parameter `NaturalAction` may be set to `true`, in which case the resulting group has the natural action on the basis vectors (not the coordinate vectors); note that in this case the resulting matrix group will have (non-integral) rational entries in general, even though the image under the group of an integral basis vector will always be integral.

The algorithm uses a backtrack search to find images of the basis vectors. A vector is a possible image if it has the correct inner product with the images chosen so far. Additional invariants which have to be respected by automorphisms are used by default and are usually sufficient for satisfactory performance. For difficult examples the parameters described below allow to consider further invariants which are more sophisticated and harder to compute, but often find dead ends in the backtrack at an early stage.

The algorithm computes and stores a set of short vectors in the lattice that spans the lattice and is guaranteed closed under the action of the automorphism group. This restricts its general application, as for high dimensional lattices the number of vectors of minimal length may be too large to work with. However, the function can of course be applied to lattices in higher dimensions with a reasonable number of short vectors.

Setting the parameter `Stabilizer := i` will cause the function to compute only the point stabilizer of i basis vectors. These will in general not be the first i basis vectors, as the function chooses the basis to speed up the computation.

The parameter `Depth` is retained for compatibility with previous versions of Magma(which used Souvignier's AUTO program) but it is now ignored. The improved backtrack search achieved by using ordered partition methods has made the `Depth` concept unnecessary.

In some hard examples one may want to use Bacher polynomials, which are a combinatorial invariant that usually separates the orbits of the automorphism group on the short vectors. However, these are expensive to calculate and should only be used if one suspects that the automorphism group has many orbits on the short vectors. The parameter `BacherDepth` specifies to which depth the Bacher polynomials may used and should usually be chosen to be 1, since even small automorphism groups will have only very few (most likely 1) orbits on the vectors having correct scalar product with the first image. Setting `BacherDepth` to zero forbids the use of Bacher polynomial invariant. Setting it to anything else allows the algorithm to use this invariant at level 1. Bacher polynomials are computed by counting pairs of vectors having a certain scalar product with other vectors. This scalar product is by default chosen to be half the norm of the vector, since this will usually be the value which occurs least frequent.

In some situations one may already know a subgroup of the full automorphism group, either by the construction of the lattice or an earlier stabilizer computation. This subgroup can be made available by setting `Generators := Q`, where Q is a set or sequence containing the generators of the subgroup.

Since V2.13, the algorithm has had the ability to first attempt to compute an orthogonal decomposition of L by considering the sublattices spanned by the set of shortest vectors, and if there is a non-trivial decomposition the automorphism group is computed via an algorithm of G. Nebe which computes the automorphism groups for the components and combines these. This decomposition method can be invoked by setting `Decomposition` to `true`.

When decomposition is suppressed, it is possible to supply the backtrack algorithm with the set of vectors to be used as domain for the search. To do this set the parameter `Vectors` to be a matrix so that the rows of the matrix give the lattice elements to be used. (Only one of a vector and its negative should be given.) To guarantee correctness of the result, the rows of the matrix should satisfy two conditions:

The rows, together with their negations, must be closed under the action of the full automorphism group, and

The sublattice of L generated by the rows of the matrix must equal L.

These conditions are not checked by the code, and it is up to the user of this parameter to ensure the correctness of their input. For example, the function `ShortVectorsMatrix` returns a matrix which satisfies the first condition. If the first condition is not satisfied, there are no guarantees about what will happen. The second condition may be violated and still give a useful result. In particular, if the sublattice generated by the vectors given has finite index in the full lattice, then the final result will be the correct automorphism group. The problem is that the backtrack search may not be particularly efficient. This may still be better than working with a very large set of vectors satisfying the second condition.

`AutomorphismGroup(L, F)`

`Stabilizer`	RngIntElt	*Default :* 0
`BacherDepth`	RngIntElt	*Default :* 0
`Generators`	[GrpMatElt]	*Default :*
`Vectors`	Mtrx	*Default :*

This function computes the subgroup of the automorphism group of the lattice L which fixes also the forms given in the set or sequence F. The matrices in F are not required to be positive definite or even symmetric. This is for example useful to compute automorphism groups of lattices over algebraic number fields. The parameters are as above.

`AutomorphismGroup(F)`

`Stabilizer`	RngIntElt	*Default :* 0
`BacherDepth`	RngIntElt	*Default :* 0
`Generators`	[GrpMatElt]	*Default :*

This function computes the matrix group fixing all forms given as matrices in the sequence F. The first form in F must be symmetric and positive definite, while the others are arbitrary. The call of this function is equivalent to `AutomorphismGroup( LatticeWithGram(F[1]), [ F[i] : i in [2..#F] ] )`. This function can be used to compute the Bravais group of a matrix group G which is defined to be the full automorphism group of the forms fixed by G. The parameters are as above.

Example H31E1

In this example 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. Note that this exactly the same as using the `NaturalAction` parameter for the function `AutomorphismGroup`.

```
> 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 ]
> AutomorphismGroup(L: NaturalAction) eq A;
true
```

Example H31E2

In this example we demonstrate how stabilizers can be used to obtain a big subgroup of the full automorphism group fairly quickly.

```
> L := Lattice("Kappa", 13);
> LL, T := PairReduce(L);
> time G2 := AutomorphismGroup(L : Stabilizer := 2);
Time: 0.030
> #G2;
48
> time GG2 := AutomorphismGroup(LL : Stabilizer := 2);
Time: 0.020
> #GG2;
48
> Z := IntegerRing();
> H := MatrixGroup< 13, Z | G2, [ T^-1*g*T : g in Generators(GG2) ] >;
> #H;
311040
> MatrixRing(Z, 13) ! -1 in H;
false
```

The pair reduction yields a different basis for the same lattice and both 2-point stabilizers have order 48. After transforming the automorphisms of LL back to automorphisms of L, the two stabilizers generate a group of reasonable size which still can be enlarged by an index of 2 by adding $-I$. We thus obtain a group of order 622080 which turns out to be the full automorphism group of the lattice L.

```
> time G := AutomorphismGroup(L);
Time: 0.020
> #G;
622080
```

Example H31E3

```
> L := Lattice("Lambda", 19);
> time G := AutomorphismGroup(L);
Time: 0.300
> #G;
23592960
> DS := DerivedSeries(G);
> [ #DS[i]/#DS[i+1] : i in [1..#DS-1] ];
[ 4, 3, 4 ]
```

```
> [ IsElementaryAbelian(DS[i]/DS[i+1]) : i in [1..#DS-1] ];
[ true, true, true ]
> H := DS[#DS];
> C := Core(G, Sylow(H, 2));
> Q := H/C; #Q, IsSimple(Q);
60 true
> LS := LowerCentralSeries(C);
> [ #LS[i]/#LS[i+1] : i in [1..#LS-1] ];
[ 256, 16, 2 ]
```

Hence, $G := \mathrm{Aut}(\Lambda_{19})$ has a series of normal subgroups with factors $2^2, 3, 2^2, A_5, 2^8, 2^4, 2$.

`IsIsometric(L, M)`

`IsIsomorphic(L, M)`

`BacherDepth`	RngIntElt	*Default :* 0
`LeftGenerators`	[GrpMatElt]	*Default :*
`RightGenerators`	[GrpMatElt]	*Default :*
`LeftVectors`	Mtrx	*Default :*
`RightVectors`	Mtrx	*Default :*

This function determines whether the lattices L and M are isometric. The method is a backtrack search analogous to the one used to compute the automorphism group of a lattice. If the lattices are isometric, the function returns a transformation matrix T as a second return value such that $F_2 = TF_1T^{tr}$, where F_1 and F_2 are the Gram matrices of L and M, respectively.

For isometric lattices the cost of finding an isometry is roughly the cost of finding one automorphism of the lattice. Again, the computation may be sped up by using the additional invariants described for the automorphism group computation.

In many applications one will check whether a lattice is isometric to one for which the automorphism group is already known. In this situation the automorphism group of the second lattice can be made available by setting `RightGenerators := Q`, where Q is a set or sequence containing the generators of the group. Note however, that for isometric lattices this may slow down the computation, since generators for stabilizers have to be recomputed. Similarly, generators for an automorphism group of the first lattice may be supplied as `LeftGenerators`.

Corresponding to the `Vectors` parameter for automorphism group calculation, the parameters `LeftVectors` and `RightVectors` allow the user to. As above, left refers to the first lattice, right to the second. Either both of these parameters can be set, or neither, an error results if just one is set. The restrictions on what constitutes correct values for these parameters are as for the `Vectors` parameter above. For correct isometry testing, the vectors given must be such that any isometry will map the left vectors into the union of the right vectors and their negatives. These conditions are not checked in the code, they are the responsibility of the user.

`IsIsometric(L, F_1, M, F_2)`

`IsIsomorphic(L, F_1, M, F_2)`

`IsIsometric(L, M)`

`IsIsomorphic(L, M)`

BacherDepth	RngIntElt	*Default :* 0
LeftGenerators	[GrpMatElt]	*Default :*
RightGenerators	[GrpMatElt]	*Default :*
LeftVectors	Mtrx	*Default :*
RightVectors	Mtrx	*Default :*

This function determines whether the lattices L and M are isometric with an isometry respecting also additional bilinear forms given by the sequences of Gram matrices F_1 and F_2. The return values and parameters are as above.

`IsIsometric(F_1, F_2)`

`IsIsomorphic(F_1, F_2)`

BacherDepth	RngIntElt	*Default :* 0
LeftGenerators	[GrpMatElt]	*Default :*
RightGenerators	[GrpMatElt]	*Default :*
LeftVectors	Mtrx	*Default :*
RightVectors	Mtrx	*Default :*

For two sequences of F_1 and F_2 of Gram matrices, determine whether a simultaneous isometry exists, i.e., a matrix T such that $TF_1[i]T^{tr} = F_2[i]$ for i in $[1..\#F_1]$. The first form in both sequences must be positive definite. The return values and parameters are as above.

Example H31E4

We construct the 16-dimensional Barnes-Wall lattice in two different ways and show that the so-obtained lattices are isometric.

```
> L := Lattice("Lambda", 16);
> LL := Lattice(ReedMullerCode(1, 4), "B");
> time bool, T := IsIsometric(L, LL : Depth := 4);
Time: 2.029
> bool;
true
> T * GramMatrix(L) * Transpose(T) eq GramMatrix(LL);
true
```

We can also show that L is a 2-modular lattice (i.e., isometric to its rescaled dual).

```
> IsIsometric(L, Dual(L));
true
```

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

31.2.1 Automorphism Group and Isometry Testing over $\mathbf{F}_q[t]$

Let q be some power of an odd prime. A bilinear form b over $\mathbf{F}_q[t]$ is said to be definte if the corresponding quadratic form is anisotropic over the completion of $\mathbf{F}_q(t)$ at the infinite place $(1/t)$.

The functions in this section compute automorphism groups and isometries of definite bilinear forms over $\mathbf{F}_q[t]$.

```
DominantDiagonalForm(X)
```

`Canonical` BOOL *Default :* `false`

`ExtensionField` FLDFIN *Default :*

Let X be a symmetric $n \times n$-matrix of rank n over a polynomial ring $K[t]$ where K denotes a field of characteristic different from 2. The function returns a symmetric matrix G and some $T \in \mathrm{GL}(n, K[t])$ such that $G = TXT^{tr}$ has dominant diagonal. I.e. the degrees of the diagonal entries of G are ascending and the degree of a non-diagonal entry is less than the degrees of the corresponding diagonal entries (see [Ger03]).

If K is a finite field and X represents a definite form and `Canonical` is set to `true`, then the form G will be unique and the third return value will be the automorphism group of G i.e. the stabilizer of G in $\mathrm{GL}(n, K[t])$. The algorithm employed is [Kir12]. Note however, the uniqueness depends on some internal choices being made. Thus the fourth return value is a finite field E which must be given as the optional argument `ExtensionField` in subsequent runs over K to ensure that results are compatible (c.f. the following example). In particular, the defining polynomial and the primitive element of E are important for the uniqueness.

Example H31E5

We test whether two definite forms over $\mathbf{F}_q[t]$ are isometric.

```
> R<t> := PolynomialRing( GF(5) );
> X1:= SymmetricMatrix([ t^3, t+1, 2*t^2+2*t+2 ] );
> X2:= SymmetricMatrix([ t^3, t^4+2*t+2, t^5+2*t^2+2*t+3 ]);
> G1, T1, Aut, E:= DominantDiagonalForm(X1 : Canonical);
> T1 * X1 * Transpose(T1) eq G1;
true
> GG:= [ Matrix(g) : g in Generators(Aut) ];
> forall{g : g in GG | g * G1 * Transpose(g) eq G1 };
true
```

So the form G_1 is invariant under Aut. Now we reduce the second form X_2. To be able to compare the results, we have to provide the field E from above.

```
> G2, T2 := DominantDiagonalForm(X2 : Canonical, ExtensionField:= E);
> G1 eq G2;
true
```

Thus the two forms X_1 and X_2 are isometric and $T_1^{-1}T_2$ is an isometry.

`AutomorphismGroup(G)`

`ExtensionField` FldFin *Default :*

Computes the automorphism group of the definite bilinear form given by the symmetric matrix G over $\mathbf{F}_q[t]$.

The second return value is a finite field as explained in `DominantDiagonalForm` above. It may be supplied for later calls over the same ground field $\mathbf{F}_q$ via the optional argument `ExtensionField` to safe some time if q is large. The correctness of the algorithm does not depend on it.

`IsIsometric(G1, G2)`

`ExtensionField` FldFin *Default :*

Tests whether two definite bilinear forms over $\mathbf{F}_q[t]$ are isometric. If so, the second return value is a matrix $T \in \mathrm{GL}(n, q)$ such that $TG1T^{tr} = G2$.

The third return value is a finite field as explained in `DominantDiagonalForm` above. It may be supplied for later calls over the same ground field $\mathbf{F}_q$ via the optional argument `ExtensionField` to safe some time if q is large. The correctness of the algorithm does not depend on it.

`ShortestVectors(G)`

Returns a sequence Q which contains the shortest non-zero vectors with respect to a given definite bilinear form G over $\mathbf{F}_q[t]$ where q is odd. The sequence Q contains tuples $< v, r >$ where v is a shortest vector and r denotes its norm with respect to G.

`ShortVectors(G, B)`

Let G be a definite bilinear form of rank n over $\mathbf{F}_q[t]$ for some odd q. The function returns a sequence Q which contains all vectors in $\mathbf{F}_q[t]^n$ whose norm with respect to G is at most B. The sequence Q contains tuples $< v, r >$ where v is such a short vector and r denotes its norm with respect to G.

31.3 Lattices from Matrix Groups

In Magma a G-lattice L is a lattice upon which a finite integral matrix group G acts by right multiplication.

Each G-lattice L has references to both the original ("natural") group G which acts on the standard lattice in which L is embedded and also the reduced group of L which is the reduced representation of G on the basis of L.

31.3.1 Creation of G-Lattices

The following functions create G-lattices. Note that the group G must be a finite integral matrix group.

`Lattice(G)`

Given a finite integral matrix group G, return the standard G-lattice (with standard basis and rank equal to the degree of G).

`LatticeWithBasis(G, B)`

Given a finite integral matrix group G and a non-singular matrix B whose row space is invariant under G (i.e., $Bg = T_gB$ for each $g \in G$ where T_g is a unimodular integral matrix depending on g), return the G-lattice with basis matrix B. (The number of columns of B must equal the degree of G; G acts naturally on the lattice spanned by B.)

`LatticeWithBasis(G, B, M)`

Given a finite integral matrix group G, a non-singular matrix B whose row space is invariant under G (i.e., $Bg = T_gB$ for all $g \in G$ where T_g is a unimodular integral matrix depending on g) and a positive definite matrix M invariant under G (i.e., $gMg^{tr} = M$ for all $g \in G$) return the G-lattice with basis matrix B and inner product matrix M. (The number of columns of B must equal the degree of G and both the number of rows and the number of columns of M must equal the degree of G; G acts naturally on the lattice spanned by B and fixes the Gram matrix of the lattice).

`LatticeWithGram(G, F)`

Given a finite integral matrix group G and a positive definite matrix F invariant under G (i.e., $gFg^{tr} = F$ for all $g \in G$) return the G-lattice with standard basis and inner product matrix F (and thus Gram matrix F). (Both the number of rows and the number of columns of M must equal the degree of G; G fixes the Gram matrix of the returned lattice).

31.3.2 Operations on G-Lattices

The following functions provide basic operations on G-lattices.

```
IsGLattice(L)
```

Given a lattice L, return whether L is a G-lattice (i.e., there is a group associated with L).

```
Group(L)
```

Given a G-lattice L, return the matrix group of the (reduced) action of G on L. The resulting group thus acts on the coordinate lattice of L (like the automorphism group).

```
NumberOfActionGenerators(L)
Nagens(L)
```

Given a G-lattice L, return the number of generators of G.

```
ActionGenerator(L, i)
```

Given a G-lattice L, return the i-th generator of the (reduced) action of G on L. This is the reduced action of the i-th generator of the original group G (which may be the identity matrix).

```
NaturalGroup(L)
```

Given a G-lattice L, return the matrix group of the (natural) action of G on L. The resulting group thus acts on L naturally.

```
NaturalActionGenerator(L, i)
```

Given a G-lattice L, return the i-th generator of the natural action of G on L. This is simply the i-th generator of the original group G.

31.3.3 Invariant Forms

The functions in this section compute invariant forms for G-lattices.

```
InvariantForms(L)
```

For a G-lattice L, return a basis for the space of invariant bilinear forms for G (represented by their Gram matrices) as a sequence of matrices. The first entry of the sequence is a positive definite symmetric form for G.

```
InvariantForms(L, n)
```

For a G-lattice L, return a sequence consisting of $n \geq 0$ invariant bilinear forms for G.

```
SymmetricForms(L)
```

For a G-lattice L, return a basis for the space of symmetric invariant bilinear forms for G. The first entry of the sequence is a positive definite symmetric form of G.

`SymmetricForms(L, n)`

For a G-lattice L, return a sequence of $n \geq 0$ independent symmetric invariant bilinear forms for G. The first entry of the first sequence (if $n > 0$) is a positive definite symmetric form for G.

`AntisymmetricForms(L)`

For a G-lattice L, return a basis for the space of antisymmetric invariant bilinear forms for G.

`AntisymmetricForms(L, n)`

For a G-lattice L, return a sequence of $n \geq 0$ independent antisymmetric invariant bilinear forms for G.

`NumberOfInvariantForms(L)`

For a G-lattice L, return the dimension of the space of (symmetric and antisymmetric) invariant bilinear forms for G. The algorithm uses a modular method which is always correct and is faster than the actual computation of the forms.

`NumberOfSymmetricForms(L)`

For a G-lattice L, return the dimension of the space of symmetric invariant bilinear forms for G.

`NumberOfAntisymmetricForms(L)`

For a G-lattice L, return the dimension of the space of antisymmetric invariant bilinear forms for G.

`PositiveDefiniteForm(L)`

For a G-lattice L, return a positive definite symmetric form for G. This is a positive definite matrix F such that $gFg^{tr} = F$ for all $g \in G$.

31.3.4 Endomorphisms

The functions in this subsection compute endomorphisms of G-lattices. This is done by approximating the averaging operator over the group and applying it to random elements.

`EndomorphismRing(L)`

For a G-lattice L, return the endomorphism ring of L as a matrix algebra over $\mathbf{Q}$.

`Endomorphisms(L, n)`

For a G-lattice L, return a sequence containing n independent endomorphisms of L as elements of the corresponding matrix algebra over $\mathbf{Q}$. n must be in the range $[0..d]$, where d is the dimension of the endomorphism ring of L. This function may be useful in situations where the full endomorphism algebra is not required, e.g., to split a reducible lattice.

```
DimensionOfEndomorphismRing(L)
```

Return the dimension of the endomorphism algebra of the G-lattice L by a modular method (which always yields a correct answer).

```
CentreOfEndomorphismRing(L)
```

For a G-lattice L, return the centre of the endomorphism ring of L as a matrix algebra over $\mathbf{Q}$.

This function can be used to split a reducible lattice into its homogeneous components.

```
CentralEndomorphisms(L, n)
```

For a G-lattice L, return a sequence containing n independent central endomorphisms of L as elements of the corresponding matrix algebra over $\mathbf{Q}$. n must be in the range $[0..d]$, where d is the dimension of the centre of the endomorphism ring of L.

```
DimensionOfCentreOfEndomorphismRing(L)
```

Return the dimension of the centre of the endomorphism algebra of the G-lattice L by a modular method (which always yields a correct answer).

31.3.5 G-invariant Sublattices

The functions in this section compute G-invariant sublattices of a given G-lattice L.

For a fixed prime p, the algorithm constructs the maximal G-invariant sublattices of L as kernels of $\mathbf{F}_pG$-epimorphisms $L/pL \to S$ for some simple $\mathbf{F}_pG$-module S as described in [Ple74].

Iterating this process yields all G-invariant sublattices of L whose index in L is a p-power. Finally, intersecting lattices of coprime index yields all sublattices of L.

```
Sublattices(G, Q)
Sublattices(L, Q)
```

`Limit`	RngIntElt	*Default :* ∞
`Levels`	RngIntElt	*Default :* ∞
`Projections`	[Mtrx]	*Default :* []

Given either

(a) an integral matrix group G with natural lattice $L = \mathbf{Z}^n$

(b) a sequence G of integral matrices generating a $\mathbf{Z}$-order in $\mathbf{Q}^{n\times n}$ with natural lattice $L = \mathbf{Z}^n$

(c) a G-lattice L in $\mathbf{Q}^n$.

together with a set or sequence Q of primes, compute the G-invariant sublattices of L (as a sequence) which are not contained in pL for any $p \in Q$ and whose index in L is a product of elements of Q.

This set of G-invariant sublattices of L is finite if and only if $\mathbf{Q}_p \otimes L$ is irreducible as a $\mathbf{Q}_pG$-module for all $p \in Q$.

Setting the parameter `Limit := n` will terminate the computation after n sublattices have been found.

Setting the parameter `Levels := n` will only compute sublattices M such that L/M has at most n composition factors.

The optional parameter `Projections` can be a sequence of n by n matrices that describe projections on $\mathbf{Q}^n$ that map L to itself. In this case, MAGMA will only compute those sublattices of L which have the same images under the projections as L does.

The second return value indicates whether the returned sequence contains all such sublattices or not.

`Sublattices(G, p)`

`Sublattices(L, p)`

`Limit`	RNGINTELT	*Default :* ∞
`Levels`	RNGINTELT	*Default :* ∞
`Projections`	[MTRX]	*Default :* []

The same as the above where the set Q consists only of the given ptime p.

`Sublattices(G)`

`Sublattices(L)`

`Limit`	RNGINTELT	*Default :* ∞
`Levels`	RNGINTELT	*Default :* ∞
`Projections`	[MTRX]	*Default :* []

For an integral matrix group G or a G-lattice L this intrinsic equals the one above with Q taken to be the prime divisors of the order of G.

`SublatticeClasses(G)`

`MaximalOrders`	BOOLELT	*Default :* `false`

For an integral matrix group G returns representatives for the isomorphism classes of G-invariant lattices (i.e. the orbits under the unit group of the endomorphism ring E of G).

If `MaximalOrders` is set to `true`, only sublattice classes which are invariant under some maximal order of E are considered.

Currently the function requires E to be a field.

Example H31E6

We construct sublattices of the standard G-lattice where G is an absolutely irreducible degree-8 integral matrix representation of the group $\mathrm{GL}(2,3) \times \mathrm{S}_3$.

We first define the group G.

```
> G := MatrixGroup<8, IntegerRing() |
>    [-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,  1,  0,  0,  0,  0,  0,  0,
>     -1,  0,  0,  0,  1,  0,  0,  0,
>      0,  0, -1,  0,  0,  0,  1,  0,
>      0,  0,  0,  1,  0,  0,  0, -1,
>      0,  1,  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, -1,  0,  0,  0,
>      0,  0,  0,  0,  0,  1,  0,  0,
>      0,  0,  0, -1,  0,  0,  0,  1,
>      0,  0, -1,  0,  0,  0,  1,  0,
>      1,  0,  0,  0, -1,  0,  0,  0,
>      0, -1,  0,  0,  0,  1,  0,  0]>;
```

We next compute the unique positive definite form F fixed by G.

```
> time F := PositiveDefiniteForm(G);
Time: 0.050
> F;
[2 0 0 0 1 0 0 0]
[0 2 0 0 0 1 0 0]
[0 0 2 0 0 0 1 0]
[0 0 0 2 0 0 0 1]
[1 0 0 0 2 0 0 0]
[0 1 0 0 0 2 0 0]
[0 0 1 0 0 0 2 0]
[0 0 0 1 0 0 0 2]
```

We now compute all sublattices of the standard G-lattice.

```
> time Sub := Sublattices(G);
Time: 0.370
> #Sub;
18
```

For each sublattice we compute the invariant positive definite form for the group given by the action of G on the sublattice.

```
> PrimitiveMatrix := func<X |
>     P ! ((ChangeRing(P, RationalField()) ! X) / GCD(Eltseq(X)))
>         where P is Parent(X)>;
> FF := [PrimitiveMatrix(B * F * Transpose(B))
>             where B is BasisMatrix(L): L in Sub];
```

We next create the sequence of all the lattices whose Gram matrices are given by the (LLL-reduced) forms.

```
> Sub := [LatticeWithGram(LLLGram(F)) : F in FF];
> #Sub;
18
```

We now compute representatives for the **Z**-isomorphism classes of the sequence of lattices.

```
> Rep := [];
> for L in Sub do
>     if forall{LL: LL in Rep | not IsIsometric(L, LL)} then
>         Append(~Rep, L);
>     end if;
> end for;
> #Rep;
4
```

Thus there are 4 non-isomorphic sublattices. We note the size of the automorphism group, the determinant, the minimum and the kissing number of each lattice. (In fact, the automorphism groups of these 4 lattices happen to be maximal finite subgroups of $\mathrm{GL}(8, \mathbf{Q})$ and all have $\mathrm{GL}(2, 3) \times \mathrm{S}_3$ as a common irreducible subgroup.)

```
> time A := [AutomorphismGroup(L) : L in Rep];
Time: 0.240
> [#G: G in A];
[ 497664, 6912, 696729600, 2654208 ]
> [Determinant(L): L in Rep];
[ 81, 1296, 1, 16 ]
> [Minimum(L): L in Rep];
[ 2, 4, 2, 2 ]
> [KissingNumber(L): L in Rep];
[ 24, 72, 240, 48 ]
```

Finally, we note that each lattice is isomorphic to a standard construction based on root lattices.

```
> l := IsIsometric(Rep[1],
>        TensorProduct(Lattice("A", 2), StandardLattice(4))); l;
true
> l := IsIsometric(Rep[2],
>        TensorProduct(Lattice("A", 2), Lattice("F", 4))); l;
true
> l := IsIsometric(Rep[3], Lattice("E", 8)); l;
true
> l := IsIsometric(Rep[4],
>        TensorProduct(Lattice("F", 4), StandardLattice(2))); l;
```

Example H31E7

This example illustrates the optional argument `Projections`.

```
> G := MatrixGroup<4, IntegerRing() |
> [ -1, 0, 1, 0, 0, -1, 1, -3, -1, 0, 0, 0, 0, 0, 0, 1 ],
> [ -1, 0, 0, 0, -3, 2, 0, 3, 0, 0, -1, 0, 1, -1, 0, -1 ] >;
> E := EndomorphismRing(G);
> I := CentralIdempotents(ChangeRing(E, RationalField())); I;
[
    [ 0  0  0  0]
    [-1  1  0  0]
    [ 0  0  0  0]
    [ 0  0  0  1],
    [1 0 0 0]
    [1 0 0 0]
    [0 0 1 0]
    [0 0 0 0]
]
```

Since the central idempotents are all integral, they map the standard lattice $\mathbf{Z}^n$ to itself. Even though this group G fixes infinitely many sublattices of Z^n (even up to scalar multiples), there can only be finitely many which have the same images under the central idempotents as $\mathbf{Z}^n$.

```
> S := Sublattices(G : Projections:= I); #S;
3
```

So in this case there are only three such lattices. To check that the lattices do project correctly, we can use

```
> I := [ Matrix(Integers(), i) : i in I ];
> Images := [ [Image(BasisMatrix(s) * i) : i in I] : s in S ];
> #Set(Images) eq 1;
true
```

31.3.6 Lattice of Sublattices

Magma can construct the lattice V of all G-invariant sublattices of the standard lattice $L = \mathbf{Z}^n$. Various properties of the lattice V may then be examined. Magma only stores the primitive sublattices of L, i.e. those sublattices that are not contained in kL for some $k > 1$.

In general, G fixes infinitely many primitive lattices. Thus one has to limit the number of sublattices to be constructed just as in the `Sublattice` intrinsic. In this case, all operations on V like coercions, intersections, sums etc. assume that the result of the operation is again is a scalar multiple of some element stored in V.

The lattice V has type `LatLat` and elements of V have type `LatLatElt` and are numbered from 1 to n where n is the number of primitive sublattices of L that have been constructed in the beginning.

31.3.6.1 Creating the lattice of sublattices

`SublatticeLattice(G, Q)`

`Limit`	RngIntElt	*Default :* ∞
`Levels`	RngIntElt	*Default :* ∞
`Projections`	[Mtrx]	*Default :* []

Given either an integral matrix group G of degree n or a sequence G of integral matrices generating a $\mathbf{Z}$-order in $\mathbf{Q}^{n\times n}$ together with a set or sequence Q of primes, compute the G-invariant sublattices of $\mathbf{Z}^n$ (as a sequence) which are not contained in $p\mathbf{Z}^n$ for any $p \in Q$ and whose index in $\mathbf{Z}^n$ is a product of elements of Q.

The second return value indicates whether all G-invariant lattices have been constructed.

The optional parameters are the same as for the `Sublattices` intrinsic.

`SublatticeLattice(G, p)`

`Limit`	RngIntElt	*Default :* ∞
`Levels`	RngIntElt	*Default :* ∞
`Projections`	[Mtrx]	*Default :* []

Same as above where the set Q consists only of the given prime p.

`SublatticeLattice(G)`

`Limit`	RngIntElt	*Default :* ∞
`Levels`	RngIntElt	*Default :* ∞
`Projections`	[Mtrx]	*Default :* []

Same as above where the set Q is taken to be set of prime divisors of the order of the group G.

Example H31E8

This example shows how to create a lattice of sublattices.

```
> G:= sub< GL(2, Integers()) | [0,1,-1,0] >;
> V:= SublatticeLattice(G); V;
Lattice of 2 sublattices
```

31.3.6.2 Operations on the Lattice of Sublattices

In the following, V is a lattice of G-invariant lattices for some group or **Z**-order G and Q denotes the set of primes that where used to create V.

`#V`

The number of (primitive) lattices stored in V.

`V ! i`

The i-th element of the lattice V with respect to the internal labeling.

`V ! M`

Given a (basis matrix of some) G-invariant lattice M, create the element of the lattice V corresponding to M.

`NumberOfLevels( V )`

The number of different levels (layers) stored in V. Note that levels are counted starting from 0.

`Level(V, i)`

The primitive lattices stored at the i-th level (layer). Note that levels are counted starting from 0.

`Levels(v)`

The i-th entry of the result is a sequence of the primitive lattice elements lying on the $i-1$-th level.

`Primes(V)`

The primes that where used to create V.

`Constituents(V)`

A sequence containing the constituents (simple $\mathbf{F}_pG$ - modules) that where used during the construction of the G-lattices in V.

`IntegerRing() ! e`

The integer corresponding to lattice element e.

`e + f`

The sum of the lattice elements e and f.

`e meet f`

The intersection of the lattice elements e and f.

`e eq f`

Tests whether e and f are equal.

`MaximalSublattices(e)`

The sequence S of maximal sublattices of e having index p for some $p \in Q$. The second return value is a list C of integers such that $S[i]/e$ is isomorphic to the $C[i]$-th constituent of V. The ordering of the constituents is the same as in the `Constituents` intrinsic.

`MinimalSuperlattices(e)`

The sequence S of minimal superlattices of e in which e has index p for some $p \in Q$. The second return value is a list C of integers such that $e/S[i]$ is isomorphic to the $C[i]$-th constituent of V. The ordering of the constituents is the same as in the `Constituents` intrinsic.

`Lattice(e)`

The G-lattice corresponding to e.

`BasisMatrix(e)`

`Morphism(e)`

The basis matrix of the G-lattice corresponding to e.

Example H31E9

Let G be the automorphism group of the root lattice A_5. Since G is absolutely irreducible, it fixes only finitely many lattices up to scalars. We explore them.

```
> G:= AutomorphismGroup(Lattice("A", 5));
> FactoredOrder(G);
[ <2, 5>, <3, 2>, <5, 1> ]
> #SublatticeLattice(G, 5);
1
```

Hence there are no primitive sublattices between L and $5L$. Hence it suffices to check only the lattices at 2 and 3 the two remaining prime divisors of the order of G.

```
> V:= SublatticeLattice(G, {2,3}); #V;
4
> M:= MaximalSublattices(V ! 1); M;
[
    sublattice number 2,
    sublattice number 3
]
> V ! 2 meet V ! 3;
sublattice number 4
```

Moreover, the second and third lattice are (up to rescaling) dual to each other with respect to some G-invariant form.

```
> F:= PositiveDefiniteForm(G);
> L:= Dual(Lattice(BasisMatrix(V ! 2), F) : Rescale:= false);
> V ! L;
```

```
sublattice number 3 times 1/6
```

In particular, every G-invariant lattice can be constructed from lattice number 2 by taking scalar multiples, duals, sums and intersections. For example the standard lattice can be written as:

```
> (V ! 2) + (V ! (6*L));
sublattice number 1
```

Example H31E10

Let G be the 8-dimensional (faithful) rational representation of $\mathrm{SL}(2, 7)$. Its endomorphism ring E is isomorphic to $\mathbf{Q}(\sqrt{-7})$. We find all G-invariant lattices of G that are invariant under the maximal order M of E up to multiplication with elements in E. After this is done, we quickly obtain all finite subgroups of $\mathrm{GL}(8, \mathbf{Q})$ (up to conjugacy) that include a normal subgroup conjugate to G.
To shorten the example, we choose G such that the standard lattice L is already invariant under M.

```
> SetSeed(1);
> G:= MatrixGroup<8, IntegerRing() |
>    [ 0, 1, 0, 0, 0, 0, -1, 0, -1, 0, 0, 0, 0, 0, -1, 1,
>      0, 0, 0, 1, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, -1, 1,
>      0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, -1, 0, -1, 1,
>      0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 0, 0, 0, 0, -2, 1 ],
>    [ 0, -1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0,
>     -1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, -1, 1,
>      0, 0, 0, -1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
>      0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ] >; #G;
336
> E:= EndomorphismRing(G);
> M:= MaximalOrder(ChangeRing(E, RationalField()));
> ok, M:= CanChangeUniverse(Basis(M), MatrixRing(Integers(), 8)); ok;
true
```

So L is M-invariant. The lattices at the primes 3 and 7 are multiples of L as we can see as follows:

```
> w7:= IntegralMatrix(E.2 - Trace(E.2)/8);
> w7 div:= GCD( Eltseq(w7) );                    // a square root of 7
> V:= SublatticeLattice([ Matrix(G.i) : i in [1..Ngens(G)] ] cat M, [3,7]); #V;
2
> V ! w7;
sublattice number 2
```

So it remains to check the lattices at 2. The two prime ideals in M over 2 are generated by `p` and `q` where

```
> p:= 1 - (w7+1) div 2;
> q:= (w7+1) div 2;
> Gens:= [ Matrix(G.i) : i in [1..Ngens(G)] ];
> V:= SublatticeLattice(Gens cat M, 2: Levels:= 3);
> Levels(V);
```

```
[
    [
        sublattice number 1
    ],
    [
        sublattice number 2,
        sublattice number 3
    ],
    [
        sublattice number 4,
        sublattice number 5,
        sublattice number 6
    ],
    [
        sublattice number 7,
        sublattice number 8,
        sublattice number 9,
        sublattice number 10
    ]
]
> [ V | BasisMatrix(V ! i)*x : i in [1..3], x in [p,q] ];
[
    sublattice number 4,
    sublattice number 7,
    sublattice number 8,
    sublattice number 6,
    sublattice number 9,
    sublattice number 10
]
```

So the lattice numbers $1, 2, 3$ and 5 represent the orbits of the action of E on the set of all MG-invariant lattices. Moreover, every matrix group N normalizing G acts on the MG-invariant lattices and (up to conjugacy) thus fixes one of these four lattices. If it fixes L, it also fixes $V!2 + V!3 = V!5$ and vice versa. Similarly, it fixes $V!2$ if and only if it fixes $V!3$.

```
> F:= PositiveDefiniteForm(G);
> N1:= Normalizer(AutomorphismGroup(LatticeWithGram(F)), G); #N1;
672
> A:= AutomorphismGroup(Lattice(BasisMatrix(V ! 2), F) : NaturalAction);
> N2:= Normalizer(A, ChangeRing(G, Rationals())); #N2;
336
```

So `N1` (which is isomorphic to $2.L(2,7):2$) is up to conjugacy the only proper finite extension of G in $\mathrm{GL}(8, \mathbf{Q})$.

31.4 Bibliography

[**Ger03**] Larry J. Gerstein. Definite quadratic forms over $\mathbf{F}_q[X]$. *J. Algebra*, 268(1):252–263, 2003.

[**Kir12**] M. Kirschmer. A normal form for definite quadratic forms over $\mathbf{F}_q[t]$. *Math. Comp.*, 81:1619–1634, 2012.

[**Ple74**] Wilhelm Plesken. *Beiträge zur Bestimmung der endlichen irreduziblen Untergruppen von GL(n,Z) und ihrer ganzzahligen Darstellungen.* PhD thesis, RWTH Aachen, 1974.

[**PS97**] Wilhelm Plesken and Bernd Souvignier. Computing Isometries of Lattices. *J. Symbolic Comp.*, 24(3):327–334, 1997.

32 QUADRATIC FORMS

Chapter 32
QUADRATIC FORMS

32.1 Introduction

This chapter describes miscellaneous functionality for fairly general quadratic forms. The main feature currently here is an implementation of Simon's algorithm for finding isotropic subspaces of integral forms.

In Magma, quadratic forms are generally represented either as multivariate polynomials, or as symmetric matrices.

32.2 Constructions and Conversions

`SymmetricMatrix(f)`

Given a multivariate polynomial that is homogeneous of degree 2, this returns a symmetric matrix representing the same quadratic form.

`GramMatrix(L)`

The symmetric matrix giving the quadratic form on the lattice L.

`QuadraticForm(L)`

The quadratic form associated to the lattice L, as a multivariate polynomial.

`QuadraticForm(M)`

The quadratic form for a symmetric matrix M, as a multivariate polynomial.

32.3 Local Invariants

These commands calculate the standard invariants that characterize a quadratic form over the rationals. Definitions of the invariants may be found in Conway-Sloane [JC98], Chapter 15, Section 5.1.

```
pSignature(f,p)
pSignature(M,p)
pSignature(L,p)
```

The p-signature of the specified quadratic form over the rationals, where p is a prime number or -1 (designating the real place).

For odd primes p, this is defined by diagonalizing the form, and adding p-parts of these entries to 4 times the number of anti-squares (mod p) amongst these entries. The term "anti-square" modulo p denotes something that has: odd valuation at p; and the prime-to-p part, called u, has Kronecker symbol $(\frac{u}{p}) = -1$.

At $p = 2$ it is the sum of the odd parts of the diagonalized entries plus 4 times the number of anti-squares. In either case, the final answer is really only defined modulo 8 (this is so that p-signatures are invariant under rational equivalence).

At the real place, it is the difference between the number of positive and negative eigenvalues (the terminology here can be murky).

```
Oddity(f)
Oddity(L)
Oddity(M)
```

This returns the 2-signature of the given quadratic form over the rationals.

```
pExcess(f, p)
pExcess(M, p)
pExcess(L, p)
```

The p-excess of the specified quadratic form over the rationals, where p is a prime number or -1 (designating the real place). The p-excess is the difference between the p-signature and dimension for odd primes (including -1), and is the negation of this for $p = 2$. The sum of p-excesses over all primes should be 0 modulo 8.

```
WittInvariant(f, p)
WittInvariant(M, p)
WittInvariant(L, p)
HasseMinkowskiInvariant(f, p)
HasseMinkowskiInvariant(M, p)
HasseMinkowskiInvariant(L, p)
```

Calculates the Witt invariant (sometimes called the Hasse-Minkowski invariant) over $\mathbf{Q}_p$ of the given quadratic form. Again the form must be defined over either the rationals or the integers. The result is returned as something in the set $\{-1, +1\}$. One definition of this invariant is to diagonalize the form and then take the product (in our multiplicative notation) of the Hilbert symbols of the $\binom{n}{2}$ pairs of distinct nonzero diagonal entries, as in §5.3 of Chapter 15 of Conway-Sloane [JC98], which is what is implemented here. Another method would be to use a comparison of p-excesses with the standard form (also in Conway-Sloane). Starting with a p-adic input could lead to precision problems at the diagonalization step, and so is not allowed. Can also be called via `HasseMinkowskiInvariant`.

```
WittInvariants(f)
WittInvariants(M)
WittInvariants(L)
HasseMinkowskiInvariants(f)
HasseMinkowskiInvariants(M)
HasseMinkowskiInvariants(L)
```

Compute `WittInvariant(f,p)` for all bad primes p, and return the result of a sequence of tuples, each entry given by $\langle p, W_p(f)\rangle$. The set of bad primes includes the real place, the prime $p = 2$, and all primes that divide either the numerator or the denominator of the determinant of symmetric matrix associated to f. Can also be called via `HasseMinkowskiInvariants`.

32.4 Isotropic Subspaces

```
IsotropicSubspace(f)
IsotropicSubspace(M)
```

This returns an isotropic subspace for the given quadratic form (which must be either integral or rational), which may be given either as a multivariate polynomial f or as a symmetric matrix M. The subspace returned is in many cases guaranteed to be a maximal totally isotropic subspace. More precisely (assuming that the form is nonsingular), upon writing (r, s) for the signature of f with $r \geq s$, the dimension of the space returned is at least $\min(r, s + 2) - 2$, which is maximum possible when $s \leq r + 2$. Since version 2.18, an improvement to the original algorithm has been appended, which typically (subject to a solvability criterion for a 4-dimensional subspace) enlarges the dimension of space by 1 when $r + s$ is even and $s \leq r + 2$.

The algorithm used is due to Simon (see [Sim05]), and uses on the Bosma-Stevenhagen algorithm for the 2-part of the class groups of a quadratic field (see [BS96]). There is no corresponding intrinsic for a lattice: since the associated form is definite, there are no isotropic vectors.

Example H32E1

```
> v :=  [ 6, -2, -7, 5, -2, -10, -3, 5, -7, -3, 10, -8, 5, 5, -8, 0 ];
> M := Matrix(4, 4, v); M;
[  6  -2  -7   5]
[ -2 -10  -3   5]
[ -7  -3  10  -8]
[  5   5  -8   0]
> Determinant(M); Factorization(Determinant(M));
1936
[ <2, 4>, <11, 2> ]
> WittInvariant(M, 2);
-1
> WittInvariant(M, 11);
1
> D := Diagonalization(M); D; // signature (2,2)
[     6      0      0      0]
[     0    -96      0      0]
[     0      0     18      0]
[     0      0      0 -34848]
> pSignature(M, -1); // should be the difference of 2 and 2
0
> n := Degree(Parent(M));
> Q := Rationals();
> E := [ Q ! D[i][i] : i in [1..n]];
> &*[ &*[ HilbertSymbol(E[i], E[j], 2) : i in [j+1..n]] : j in [1..n-1]];
-1
> &*[ &*[ HilbertSymbol(E[i], E[j], 11) : i in [j+1..n]] : j in [1..n-1]];
1
> IsotropicSubspace(M);
RSpace of degree 4, dimension 2 over Integer Ring
Generators:
( 3 -3  0  4)
( 4 -4  0 -2)
Echelonized basis:
( 1 -1  0 16)
( 0  0  0 22)
> pSignature(M, 2) mod 8;
0
> pSignature(M, 11) mod 8;
4
> pSignature(M, 5) mod 8; // equals Dimension at good primes
4
```

Example H32E2

```
> SetSeed(12345);
> n := 20;
> P := PolynomialRing(Integers(),n);
> f:=&+[&+[Random([-10..10])*P.i*P.j : i in [j..n]] : j in [1..n]];
> M := ChangeRing(2*SymmetricMatrix(f), Integers()); M;
[12 -2 -8 1 -6 0 3 -5 -6 -5 10 7 -1 -3 -7 -5 -6 -3 -3 -8]
[-2 2 10 -10 3 6 -3 -9 0 -5 7 -5 -5 -4 1 -5 1 -6 7 -8]
[-8 10 14 1 6 -8 -3 4 -2 3 -4 7 0 -8 -6 -4 -5 7 -4 8]
[1 -10 1 6 -5 0 8 1 7 -1 7 -7 6 7 -1 -9 -8 1 6 6]
[-6 3 6 -5 12 7 -3 -10 -5 -4 -6 1 -5 -9 4 -8 2 5 -4 -10]
[0 6 -8 0 7 -6 -4 3 0 2 -3 0 4 10 5 -8 6 1 -7 2]
[3 -3 -3 8 -3 -4 16 -5 -10 7 -9 9 -6 -2 0 -1 1 -4 6 2]
[-5 -9 4 1 -10 3 -5 2 -4 -3 0 -3 9 -1 3 -5 2 2 -1 -10]
[-6 0 -2 7 -5 0 -10 -4 0 -1 5 1 3 5 3 0 1 -7 5 10]
[-5 -5 3 -1 -4 2 7 -3 -1 10 -9 5 2 -4 8 6 -4 9 -3 5]
[10 7 -4 7 -6 -3 -9 0 5 -9 2 2 -7 3 8 -4 7 -3 -3 6]
[7 -5 7 -7 1 0 9 -3 1 5 2 6 -10 -6 0 -2 -3 8 2 -9]
[-1 -5 0 6 -5 4 -6 9 3 2 -7 -10 -10 7 0 8 -1 -2 9 3]
[-3 -4 -8 7 -9 10 -2 -1 5 -4 3 -6 7 -6 -2 -3 3 -9 9 6]
[-7 1 -6 -1 4 5 0 3 3 8 8 0 0 -2 2 -8 5 2 3 -4]
[-5 -5 -4 -9 -8 -8 -1 -5 0 6 -4 -2 8 -3 -8 -18 5 4 6 4]
[-6 1 -5 -8 2 6 1 2 1 -4 7 -3 -1 3 5 5 6 -3 7 -7]
[-3 -6 7 1 5 1 -4 2 -7 9 -3 8 -2 -9 2 4 -3 14 5 2]
[-3 7 -4 6 -4 -7 6 -1 5 -3 -3 2 9 9 3 6 7 5 16 -3]
[-8 -8 8 6 -10 2 2 -10 10 5 6 -9 3 6 -4 4 -7 2 -3 -4]
> D := Integers() ! Determinant(M); D;
2761320038161033227112 92
> [ <u[1], WittInvariant(f, u[1])> : u in Factorization(D)];
[ <2, -1>, <3, -1>, <7, 1>, <199, 1>, <879089, -1>, <6263690372711, -1> ]
> &+[ pExcess(f, u[1]) : u in Factorization(D) cat [<-1, 0>]] mod 8;
0
> time S := IsotropicSubspace(f);
Time: 0.640
> Dimension(S);
8
> pSignature(f, -1); // difference of 12 and 8
4
> B := Basis(S);
> InnerProduct(B[1], B[1]*M);
0
> IP := InnerProduct;
> d := Dimension(S);
> &and [&and [ IP(B[i], B[j]*M) eq 0 : i in [1..d]] : j in [1..d]];
true
```

32.5 Bibliography

[**BS96**] W. Bosma and P. Stevenhagen. On the computation of quadratic 2-class groups. *Journal de théorie des nombres de Bordeaux*, 8(2):283–313, 1996.

[**JC98**] N.J.A. Sloane J.H. Conway. *Sphere Packings, Lattices and Groups*, volume 290 of *Grundlehren der Mathematischen Wissenschaften.* Springer, New York–Berlin–Heidelberg, 3rd edition, 1998.

[**Sim05**] Denis Simon. Quadratic equations in dimensions 4, 5 and more. Preprint, URL:http://www.math.unicaen.fr/~simon/, 2005.

33 BINARY QUADRATIC FORMS

Chapter 33
BINARY QUADRATIC FORMS

33.1 Introduction

A binary quadratic form is an integral form $ax^2 + bxy + cy^2$ which is represented in Magma by a tuple $\langle a, b, c\rangle$. Binary quadratic forms play an central role in the ideal theory of quadratic fields, the classical theory of complex multiplication, and the theory of modular forms. Algorithms for binary quadratic forms provide efficient means of computing in the ideal class group of orders in a quadratic field. By using the explicit relation of definite quadratic forms with lattices with nontrivial endomorphism ring in the complex plane, one can apply modular and elliptic functions to forms, and exploit the analytic theory of complex multiplication.

The structures of quadratic forms of a given discriminant D correspond to ordered bases of ideals in an order in a quadratic number field, defined up to scaling by the rationals. A form is primitive if the coefficients a, b, and c are coprime. For negative discriminants the primitive reduced forms in this structure are in bijection with the class group of projective or invertible ideals. For positive discriminants, the reduced orbits of forms are used for this purpose. Magma holds efficient algorithms for composition, enumeration of reduced forms, class group computations, and discrete logarithms. A significant novel feature is the treatment of nonfundamental discriminants, corresponding to nonmaximal orders, and the collections of homomorphisms between different class groups coming from the inclusions of these orders.

The functionality for binary quadratic forms is rounded out with various functions for applying modular and elliptic functions to forms, and for class polynomials associated to class groups of definite forms.

33.2 Creation Functions

33.2.1 Creation of Structures

For any integer D congruent to 0 or 1 modulo 4, it is possible to create the parent structure of binary quadratic forms of discriminant D.

```
BinaryQuadraticForms(D)
```

```
QuadraticForms(D)
```

Create the structure of integral binary quadratic forms of discriminant D.

33.2.2 Creation of Forms

Binary quadratic forms may be created by coercing a triple $[a, b, c]$ of integer coefficients into the parent structure of forms of discriminant $D = b^2 - 4ac$. Other constructors are provided for constructing the group identity, prime forms, or allowing the omission of third element c of the sequence.

```
Identity(Q)
```
```
Q ! 1
```

Create the principal form in the structure Q of binary quadratic forms of discriminant D. The principal form is either $X^2 - D/4Y^2$ if $D \bmod 4$ is 0 and $X^2 + XY + (D-1)/4Y^2$ if it is 1. The principal form is a reduced form representing the identity element of the class group of Q.

```
Q ! [a, b, c]
```
```
elt< Q | a, b, c >
```
```
elt< Q | a, b >
```

Returns the binary quadratic form $aX^2 + bXY + cY^2$ in the magma of forms Q of discriminant D. Here c is determined by the solution of the equality $D = b^2 - 4ac$; if no integer c exists satisfying this, an error will occur.

```
PrimeForm(Q, p)
```

If p is a split prime or a ramified prime not dividing the conductor of the magma of quadratic forms Q, returns a quadratic form $pX^2 + bXY + cY^2$ in Q.

33.3 Basic Invariants

Structures of binary quadratic forms are defined in terms of a discriminant, and membership in a structure determined by this invariant. To aid in the construction of forms, additional elementary functions are provided to test integer inputs to determine if they define valid discriminants of quadratic forms.

```
Discriminant(f)
```

The discriminant $b^2 - 4ac$ of a quadratic form $f = aX^2 + bXY + cY^2$.

```
Discriminant(Q)
```

The discriminant of the quadratic forms belonging to the magma of quadratic forms Q.

```
IsDiscriminant(D)
```

Return `true` if the integer D is the discriminant of some quadratic form; `false` otherwise.

`FundamentalDiscriminant(D)`

The fundamental discriminant corresponding to the integer D.

`IsFundamental(D)`

`IsFundamentalDiscriminant(D)`

Return `true` if D is an integer other than 0 or 1 congruent to 0 or 1 modulo 4, which is not of the form $m^2 D_K$ for $m > 1$ and any other such integer D_K.

`Conductor(Q)`

The conductor of quadratic forms whose discriminant is that of the magma of quadratic forms Q.

33.4 Operations on Forms

33.4.1 Arithmetic

`Conjugate(f)`

Given a form $f = ax^2 + bxy + cy^2$, returns the conjugate form $ax^2 - bxy + cy^2$.

`f * g`

`Composition(f, g)`

`Al`	MonStgElt	*Default : "Gauss"*
`Reduction`	BoolElt	*Default :* `false`

Returns the composition of two binary quadratic forms f and g. The operator '`*`' returns a reduced representative of the product using a fast composition algorithm of Shanks. In contrast, the default for `Composition` is `Reduction := false`, so that one can work in the group of forms, rather in the set of class group representatives. The function `Composition` takes a further parameter `Al` which specifies whether the algorithm of Gauss or Shanks, set to `"Gauss"` by default. The algorithm of Shanks performs partial intermediate reductions, so the combination `Reduction := false` and `Al := "Shanks"` are incompatible and returns a runtime error.

`f ^ n`

`Power(f, n)`

`Al`	MonStgElt	*Default : "Gauss"*
`Reduction`	BoolElt	*Default :* `false`

Returns the n-th power of a form f. The operator '`^`' returns a reduced representative, using the fast composition algorithm of Shanks. In contrast, the default for `Power` is `Reduction := false`, so that one can work in the group of forms rather than in the class group. The function `Power` takes the further parameter `Al` in order to specify whether the algorithm of Gauss or Shanks is used, set to `"Gauss"` by default. The algorithm of Shanks performs partial intermediate reductions, so the combination `Reduction := false` and `Al := "Shanks"` are incompatible and returns a runtime error.

`Reduction(f)`

`ReducedForm(f)`

Returns a reduced quadratic form equivalent to f, and the transformation matrix.

`ReductionStep(f)`

Returns the result of applying one reduction step to the quadratic form f.

`ReductionOrbit(f)`

The cycle of reduced forms equivalent to f (and each other) where f has positive discriminant.

`Order(f)`

For a binary quadratic form f, returns its order as an element of the class group $Cl(Q)$ where Q is the parent of f.

33.4.2 Attribute Access

The coefficient sequence can be accessed as a sequence of integers, providing the inverse operation to the forms coercion constructor.

`f[i]`

The i-th coefficient of f, where $1 \leq i \leq 3$.

`Eltseq(f)`

`ElementToSequence(f)`

The sequence $[a, b, c]$ where f is the form $ax^2 + bxy + cy^2$.

33.4.3 Boolean Operations

Several boolean operators apply to quadratic forms.

`f in Q`

Return `true` if and only if f is in Q, that is f and Q have the same discriminant.

`f eq g`

Return `true` if the quadratic form f and g are equal and `false` otherwise.

`IsIdentity(f)`

Return `true` if and only if f is the principal form in its parent structure.

`IsReduced(f)`

Return `true` if the quadratic form f is reduced; `false` otherwise.

`IsEquivalent(f, g)`

Return `true` if the quadratic forms f and g reduce to the same form and `false` otherwise. If `true` and the discriminant is negative, then the transformation matrix is also returned. An error is returned if the forms are not of the same discriminant.

33.4.4 Related Structures

In addition to the Parent and Category structures of binary quadratic forms of discriminant D, the quadratic forms map to the ideals of a fixed order or discriminant D in a quadratic number field.

`Parent(f)`

`Category(Q)`

`QuadraticOrder(Q)`

Given a structure of quadratic forms of discriminant D, returns the associated order of discriminant D in a quadratic field.

`Ideal(f)`

Given a quadratic form $f = ax^2 + bxy + cy^2$, returns the ideal $(a, (-b + \sqrt{D})/2)$ in the quadratic order $\mathbf{Z}[(t + \sqrt{D})/2]$, where t equals 0 or 1.

33.5 Class Group

`ReducedForms(Q)`

Given the structure of quadratic forms of negative discriminant D, returns the sequence of all primitive reduced forms of discriminant D.

`ReducedOrbits(Q)`

Given the structure of quadratic forms of positive discriminant D, returns the sequence of all reduced orbits of primitive forms of discriminant D, as an indexed set.

`ClassNumber(Q: parameters)`

`ClassNumber(D: parameters)`

`Al`	MonStgElt	*Default :* “*Automatic*”
`FactorBasisBound`	FldReElt	*Default :* 0.1
`ProofBound`	FldReElt	*Default :* 6
`ExtraRelations`	RngIntElt	*Default :* 1

The class number of binary quadratic forms Q of discriminant D. The parameter `Al` may be supplied to select the method used to calculate the class number. The possible values are `"ReducedForms"` (enumerating all reduced forms), `"Shanks"` (using a Shanks-based algorithm in the class group), `"Sieve"` or `"NoSieve"`. The default is to use reduced form enumeration for small discriminants, the Shanks algorithm for the middle range, a class group index-calculus for large discriminants and the sieve method described in [Jac99] for very large discriminants.

The remaining parameters apply to the index-calculus computations only; for details about the parameters `FactorBasisBound`, `ProofBound` and `ExtraRelations` see the description of `ClassGroup`.

`ClassGroup(Q: *parameters*)`

`FactorBasisBound`	FldReElt	*Default :* 0.1
`ProofBound`	FldReElt	*Default :* 6
`ExtraRelations`	RngIntElt	*Default :* 1
`Al`	MonStgElt	*Default :* "*Automatic*"

The class group of the binary quadratic forms Q of discriminant D. The function also returns a map from the abelian group to the structure of quadratic forms.

Depending on the size of D, either a Shanks-based method is used ([Tes98a, BJT97]) an index calculus variant ([CDO93, HM89, Coh93]) or a sieving method. The parameters `FactorBasisBound`, `ProofBound` and `ExtraRelations` apply to the index calculus method only. This method performs in two steps: In the first step a factor basis containing prime forms of norm $< B_1$ is build. Next, one looks for generators for the full lattice of relations between the forms in the factor basis. The determinant of this lattice will be the class number and the Smith-form of the relation matrix gives the structure of the class group. Once the matrix is of full rank, the algorithm will look for `ExtraRelations` more relations to potentially decrease the discriminant.

In the second stage, for all prime forms of norm $< B_2$ it is verified that they are in the class group generated by the forms of the first step. The bounds are chosen to be $B_1 :=$ `FactorBasisBound`$\cdot \log^2 |D|$ and $B_2 :=$ `ProofBound`$\cdot \log^2 |D|$, so the result is correct under the assumption of GRH.

The final result is then checked against the Euler product over the first 30.000 primes. If the quotient becomes to large, a warning is issued. In this case one should increase the `ExtraRelations` parameter.

If the parameter `Al` is set to `"Sieve"` (regardless of the size of D) or D is larger than 10^{20} then the sieving method described in [Jac99] which uses the multiple polynomial quadratic sieve (MPQS) will be used. This is currently only proven under GRH. If the parameter `Al` is set to `"NoSieve"` then the sieving method will not be used regardless of the size of D.

`ClassGroupStructure(Q: *parameters*)`

The structure of the class group of the binary quadratic forms Q of discriminant D returned as a sequence of integers giving the abelian invariants.

For details about the parameters `Al`, `FactorBasisBound`, `ProofBound` and `Extra Relations`, see `ClassGroup`.

`AmbiguousForms(Q)`

Enumerates the ambiguous forms of negative discriminant D, where D is the discriminant of the magma of binary quadratic forms Q.

`TwoTorsionSubgroup(Q)`

The subgroup of 2–torsion elements of in the class group of Q.

Example H33E1

We give an example of some computations in the class group of a magma of quadratic forms. Elements in the class group are not really represented by single reduced forms but by cycles of equivalent reduced forms.

```
> Q<z> := QuadraticField(7537543);                    // arbitrary choice
> Q := QuadraticForms(Discriminant(Q));
> C, m := ClassGroup(Q);
> C;
Abelian Group isomorphic to Z/2 + Z/76
Defined on 2 generators
Relations:
    76*C.1 = 0
    2*C.2 = 0
>                                          // get the generators as quadratic forms:
> f := m(C.1);
> g := m(C.2);
> h := g^2;
> g, h;
<-1038,4894,1493> <-887,4340,3189>
> c := [];                           // create the cycle of forms equivalent to g^2:
> repeat
>     h := ReductionStep(h);
>     Append(~c, h);
> until h eq g^2;
> P := Parent(g);
> Identity(P) in c;
true                 // this proves that the second class has order dividing 2
> for d in Divisors(76) do
>     c := [];
>     h := f^d;
>     repeat h := ReductionStep(h); Append(~c, h);
>     until h eq f^d;
>     d, Identity(P) in c, #c;
> end for;
1 false 16
2 false 14
4 false 12                                 // the cycle lengths vary
19 false 18
38 false 16
```

```
76 true 14                          // so the true order of this class is 76
```

33.6 Class Group Coercions

The class group of a nonmaximal quadratic order R of discriminant m^2D_K, are related to the class group of the maximal order O_K of fundamental discriminant D_K by an exact sequence.

$$1 \to \frac{(O_K/mO_K)^*}{O_K^*\,(\mathbf{Z}/m\mathbf{Z})^*} \to Cl(O) \to Cl(O_K) \to 1$$

Similar maps exist between quadratic orders O_1 and O_2 in a field K, with conductors m_1 and m_2, respectively, such that $m_1 \mid m_2$. The corresponding maps on quadratic forms are implemented on quadratic forms. The homomorphism is returned as a map object, or can be called directly via the coercion operator.

`FundamentalQuotient(Q)`

The quotient homomorphism from the class group of Q to the class group of fundamental discriminant.

`QuotientMap(Q1, Q2)`

Given two structures of quadratic forms Q_1 and Q_2, such that the discriminant of Q_2 equals a square times the discriminant of Q_1, the quotient homomorphism from Q_1 to Q_2 is returned as a map object.

`Q ! f`

The ! operator applies the quotient homomorphism for automatic coercion of forms f of discriminant m^2D into the structure Q of forms of discriminant D.

33.7 Discrete Logarithms

`Log(b, x)`

The discrete logarithm of binary quadratic form x with respect to base b, or -1 if Magma can that determine no solution exists. This function exists only for negative discriminant forms. Computation is done via the Pohlig-Hellman algorithm along with a collision search subroutine (a variant of Pollard's rho method). If the user is unsure whether a solution exists, it is safest to use `Log` with a time limit (see below) to prevent an infinite loop in the collision search.

`Log(b, x, t)`

Searches for up to t seconds for the discrete logarithm of binary quadratic form x with respect to base b. This function exists only for negative discriminant forms. If Magma is able to determine no solution exists, then -1 will be returned. If no solution is found within the given time frame, then -2 will be returned. Computation is done via the Pohlig-Hellman algorithm along with a collision search subroutine (a variant of Pollard's rho method).

33.8 Elliptic and Modular Invariants

Binary quadratic forms of negative discriminant describe positive definite lattices in the complex plane, with integral-valued inner product. As such, it is possible to apply modular and elliptic functions to the form, interpreting this as an element of the upper half plane.

```
Lattice(f)
```

Given a binary quadratic form $f = ax^2 + bxy + cy^2$ of negative discriminant, returns the rank two lattice of f having Gram matrix

$$\begin{pmatrix} a & b/2 \\ b/2 & c \end{pmatrix}.$$

Note that the lattice L is the half-integral lattice such that integral representations $f(x, y) = n$ are in bijection with vectors (x, y) of norm n, which will be a rational number.

```
GramMatrix(f)
```

Returns the Gram matrix of the binary quadratic form f, which need not be of negative discriminant. The matrix will be half-integral and defined over the rationals.

```
ThetaSeries(f, n)
```

The integral theta series of the binary quadratic form f to precision n.

```
RepresentationNumber(f, n)
```

The nth representation number of the form f of negative discriminant.

```
jInvariant(f)
```

For a binary quadratic form $f = ax^2 + bxy + cy^2$ with negative discriminant, return the j–invariant of f, equal to the j–invariant of $\tau = (-b + \sqrt{b^2 - 4ac})/2a$.

```
Eisenstein(k, f)
```

Given a positive even integer $k = 2n$ and a binary quadratic form $f = ax^2 + bxy + cy^2$, return the value of the Eisenstein series $E_k(L)$ at the complex lattice $L = \langle a, (-b + \sqrt{b^2 - 4ac})/2\rangle$.

```
WeierstrassSeries(z, f)
```

Given a complex power series z with positive valuation and a binary quadratic form $f = ax^2 + bxy + cy^2$, returns the q–expansion of the Weierstrass $\wp$-function at the complex lattice $L = \langle a, (-b + \sqrt{b^2 - 4ac})/2\rangle$.

Example H33E2

```
> Q := QuadraticForms(-163);
> f := PrimeForm(Q,41);
> CC<i> := ComplexField();
> PC<z> := LaurentSeriesRing(CC);
> x := WeierstrassSeries(z,f);
> y := -Derivative(x)/2;
> A := -Eisenstein(4,f)/48;
> B := Eisenstein(6,f)/864;
> Evaluate(y^2 - (x^3 + A*x + B),1/2);
1.38460866082459688100000000 E-26 - 1.30509148119017481800000000 E-26*i
```

33.9 Class Invariants

`HilbertClassPolynomial(D)`

Given a negative discriminant D, returns the Hilbert class polynomial, defined as the minimal polynomial of $j(\tau)$, where $\mathbf{Z}[\tau]$ is an imaginary quadratic order of discriminant D.

`WeberClassPolynomial(D)`

Given a negative discriminant D congruent to 1 modulo 8, returns the Weber class polynomial, defined as the minimal polynomial of $f(\tau)$, where $\mathbf{Z}[\tau]$ is an imaginary quadratic order of discriminant D and f is a particular normalized Weber function generating the same class field as $j(\tau)$. A root $f(\tau)$ of the Weber class polynomial is an integral unit generating the ring class field related to the corresponding root $j(\tau)$ of the Hilbert class polynomial by the expression

$$j(\tau) = \frac{(f(\tau)^{24} - 16)^3}{f(\tau)^{24}},$$

where $\mathrm{GCD}(D, 3) = 1$, and

$$j(\tau) = \frac{(f(\tau)^{8} - 16)^3}{f(\tau)^{8}},$$

if 3 divides D. For further details, consult Yui and Zagier [YZ97].

33.10 Matrix Action on Forms

A matrix in $\mathrm{SL}(2, \mathbf{Z})$ acts on the right on quadratic forms by the rule

$$f(x,y)\begin{pmatrix} r & s \\ t & u \end{pmatrix} = f(rx+sy, tx+uy),$$

which is provided in MAGMA by the operator `*`.

`f * M`

The action of $\mathrm{SL}(2, \mathbf{Z})$ on forms.

33.11 Bibliography

[**BJT97**] J. Buchmann, M. J. Jacobson, Jr., and E. Teske. On Some Computational Problems in Finite Abelian Groups. *Mathematics of Computation*, 66:1663–1687, 1997.

[**CDO93**] H. Cohen, F. Diaz y Diaz, and M. Olivier. Calculs de nombres de classes et de régulateurs de corps quadratiques en temps sous-exponentiel. In *Séeminaire de Théorie des Nombres, Paris, 1990–91*, volume 108 of *Progr. Math.*, pages 35–46. Birkhäuser Boston, Boston, MA, 1993.

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.

[**HM89**] J. Hafner and K. McCurley. A rigorous subexponential algorithm for computation of class groups. *Jornal American Math. Soc.*, 2:837 – 850, 1989.

[**Jac99**] M. J. Jacobson, Jr. Applying sieving to the computation of quadratic class groups. *Math. Comp.*, 68(226):859–867, 1999.

[**Tes98**] E. Teske. A Space Efficient Algorithm for Group Structure Computation. *Mathematics of Computation*, 67:1637–1663, 1998.

[**YZ97**] N. Yui and D. Zagier. On the singular values of Weber modular functions. *Mathematics of Computation*, 66(220):1645–1662, 1997.

PART VI
GLOBAL ARITHMETIC FIELDS

34 NUMBER FIELDS

Chapter 34
NUMBER FIELDS

34.1 Introduction

The number field module in Magma is based on the Kant/Kash system (Kant-V4) [KAN97], [KAN00], developed by the group of M. Pohst in Berlin.

This chapter deals only with the basic operations possible with number fields (objects of type `FldNum`) and their elements `FldNumElt`. Apart from material covered here, there is a wealth of other functions implemented and documented in detail in chapters dealing with

* Cyclotomic fields, Chapter 36
* Quadratic fields, Chapter 35
* Orders in number fields, including ideal theory, Chapter 37
* Galois Theory, Chapter 38
* Class Field Theory, Chapter 39

Furthermore, a lot of functionality of number fields is also shared by function fields of transcendence degree 1 (Chapters 42, 41) and functionality is imported from other areas, in particular, finite fields and polynomial rings.

Number fields in Magma are finite extensions of the field $\mathbf{Q}$ of rational numbers or of another number field. Number fields of the first kind, ie. number fields that are created as extensions of $\mathbf{Q}$ are referred to as absolute fields, while extensions of number fields are called relative fields.

Number fields support extended types (Section 1.14), they can be indexed by the type of the coefficient ring: `FldNum[FldRat]` refers to an absolute extension over $\mathbf{Q}$, while `FldNum[FldNum]` refers to a relative extension.

Formally, in Magma, an object K of type `FldNum` is an algebraic extension of finite degree over a number field k or $\mathbf{Q}$. It should be thought of as constructed as a quotient ring of a univariate polynomial ring over the base field modulo some irreducible polynomial: $K = k[t]/(f(t)k[t])$ Or, the field may be constructed as a multivariate quotient: $K = k[s_1, \ldots, s_n]/(f_1(s_1), \ldots, f_n(s_n))$ where all the polynomials are univariate. However, a slightly different representation is used internally.

An important consequence of this representation as a quotient of a polynomial ring is that one cannot distinguish between e.g. $\mathbf{Q}[2^{(1/3)}]$ and $\mathbf{Q}[\zeta_3 2^{(1/3)}]$ – both of them are generated using a root of $t^3 - 2$. Therefore every non trivial extension generates a **new** object – even if the same polynomial is used repeatedly, except when the user explicitly tells Magma to check whether the polynomial has been used before. This also implies that number fields are not automatically embedded into $\mathbf{C}$ the field of complex numbers, in contrast to both quadratic and cyclotomic fields that are. Using `Conjugates` or the language of places in section 34.8 all embeddings into $\mathbf{C}$ or $\mathbf{R}$ can be found and used.

It is important to remember that **Q** is **not** a number field. However, it is possible to create an extension of degree 1 over **Q** that is a number field using, for example

Example H34E1

```
> K := ext<Rationals()|Polynomial([0,1]):DoLinearExtension>;
```

One has to distinguish between number fields with a (known) primitive element α`:= K.1` which is a zero of f and number fields where no primitive element is known. In this case α_i`:= K.i` will be a zero of f_i.

Number fields always have a 'power' basis, i.e. a basis containing only powers of the zero(s) of the defining polynomial(s) and product of those powers. This allows for example to define homomorphisms from number fields by specifying images of the generators only.

An absolute extension is always an extension of **Q**. An arbitrary number field K can always be converted into an isomorphic extension of **Q** using a constructive variant of the primitive element theorem, `AbsoluteField`. Similarly, a number field defined by multiple polynomials can be converted into a field defined by a single polynomial using `SimpleExtension`.

Likewise, if a subfield k of K is known, an isomorphic field as an extension of k can be computed.

The most important facts about the different representations are the following:

* Arithmetic is fastest in absolute simple extensions. Thus, if one wants to do lots of basic arithmetic with the elements the transformation to an absolute representation is advisable. However, typically the operations are fastest when the elements are "small" in size.
* Invariants (like `Degree`, `Discriminant`, `Norm`, `Trace` etc.) are always relative to the current representation.
* Conversions of fields tend to be time consuming thus should be avoided if possible. However, once the different field representations are computed, the conversion of elements is not too time consuming.
* Some operations and invariants can (currently) only be done for absolute representations. Essentially, these are computations involving subfields and class and unit group computations.

Number fields support only arithmetic with their elements and the computation of some invariants (`GaloisGroup`, `Subfields`, `AutomorphismGroup`). Although invariants like the class group can be computed for `FldNums` this is only a shortcut for the corresponding computations for the maximal orders so e.g. `ClassGroup(K)` is expanded to `ClassGroup(MaximalOrder(K))`.

34.2 Creation Functions

The following describes how number fields may be created. It also shows some ways of creating elements of these rings and homomorphisms from these rings into an arbitrary ring.

34.2.1 Creation of Number Fields

Algebraic Number Fields can be created in a various ways, most of which involve polynomials. The fields can be created as absolute extensions, i.e. an extension of **Q** by one or more irreducible polynomial(s), or as a relative extension which is an extension of an algebraic field by one or more polynomial(s) irreducible over that field.

`NumberField(f)`

`Check`	BoolElt	*Default :* `true`
`DoLinearExtension`	BoolElt	*Default :* `false`
`Global`	BoolElt	*Default :* `false`

Given an irreducible polynomial f of degree $n \geq 1$ over $K = \mathbf{Q}$ or some number field K, create the number field $L = K(\alpha)$ obtained by adjoining a root α of f to K.

The polynomial f is allowed to have either integer coefficients, coefficients in an order of K, coefficients from the rational field or some algebraic field K. The field K will be referred to as `CoefficientField`. If the polynomial is defined over a field and the coefficients have denominators greater than 1, an equivalent polynomial $df(x)$ is used to define L, where d is the least common multiple of the denominators of the coefficients of f.

If the optional parameter `Check` is set to `false` then the polynomial is not checked for irreducibility. This is useful when building relative extensions where factoring can be time consuming.

If `DoLinearExtension` is `true` and the degree of f is 1 a trivial extension is returned. This is an object of type `FldNum` but of degree 1. Otherwise (or by default), the coefficient field of f is returned. (This is important in situations where the number of extensions matters.) Furthermore, a degree 1 extension of **Q** is a field isomorphic to **Q**, but regarded by Magma as a number field (while **Q** itself is not, since `FldRat` is not a subtype of `FldNum`). This then supports all of the number field functions (including for instance fractional ideals) while the `Rationals()` do not. On the other hand, arithmetic will be slower.

If `Global` is `true`, then Magma checks if this polynomial is the defining polynomial of some other field created using `Global := true`. In this case, the old field will be returned.

The angle bracket notation may be used to assign the root α to an identifier e.g. `L<y> := NumberField(f)` where y will be a root of f.

```
RationalsAsNumberField()
QNF()
```

This creates a number field isomorphic to **Q**. It is equivalent to `NumberField(x-1 : DoLinearExtension)`, where x is `PolynomialRing(Rationals()).1`.

The result is a field isomorphic to **Q**, but regarded by MAGMA as a number field (while **Q** itself is not, since `FldRat` is not a subtype of `FldNum`). It therefore supports all of the number field functions, while the `Rationals()` do not. On the other hand, arithmetic will be slower.

Coercion can be used to convert to and from the `Rationals()`.

```
NumberField(s)
```

`Check`	BOOLELT	*Default :* `true`
`DoLinearExtension`	BOOLELT	*Default :* `false`
`Abs`	BOOLELT	*Default :* `false`

Let K be a possibly trivial algebraic extension of **Q**. K will be referred to as the `CoefficientField`.

Given a sequence s of nonconstant polynomials $s_1, \ldots, s_m$, that are irreducible over K, create the number field $L = K(\alpha_1, \ldots, \alpha_m)$ obtained by adjoining a root α_i of each s_i to K. The polynomials s_i are allowed to have coefficients in an order of K (or **Z**) or in K or a suitable field of fractions, but if in the latter cases denominators occur in the coefficients of s_i, an integral polynomial is used instead of s_i, as in the case of the definition of a number field by a single polynomial.

If $m > 1$ and `Abs` is `false`, a tower of extension fields

$$L_0 = K \subset L_1 = K(\alpha_m) \subset L_2 = K(\alpha_{m-1}, \alpha_m) \subset \cdots \subset L_m = K(\alpha_1, \ldots, \alpha_m) = L$$

is created, and L is a relative extension by s_1 over its ground field $L_{m-1} = K(\alpha_2, \ldots, \alpha_m)$. Thus, this construction has the same effect as m applications of the `ext` constructor. The angle bracket notation may be used to assign the m generators α_i to identifiers: `L<a`$_1$`, ..., a`$_m$`> := NumberField([ s`$_1$`, ..., s`$_m$` ]);` thus the first generator a_1, which corresponds to `L.1`, generates L over its ground field.

Note that it is important to ensure that in each of the above steps the polynomial s_i is irreducible over L_{i-1}; by default MAGMA will check that this is the case. If the optional parameter `Check` is set to `false` then this checking will not be done.

If the optional parameter `Abs` is changed to `true`, then a non-simple extension will be returned. This is a extension of the coefficient field of the f_i but such that the base field of L will be K. The ith generator will be a root of the ith polynomial in this case, but all of the generators will have L as parent. In this case, a sparse representation of number field elements will be used (based on multivariate polynomial rings). As a consequence, costs for arithmetic operations will (mainly) depend on the number of non-zero coefficients of the elements involved rather than the field degree. This allows to define and work in fields of degree $< 10^6$. However, for general elements this representation is slower than the dense (default) representation.

If the optional parameter `DoLinearExtension` is set to `true`, linear polynomials will not be removed from the list.

```
ext< F | s1, ..., sn >
```

```
ext< F | s >
```

`Check`	BoolElt	*Default :* `true`
`Global`	BoolElt	*Default :* `false`
`Abs`	BoolElt	*Default :* `false`
`DoLinearExtension`	BoolElt	*Default :* `false`

Construct the number field defined by extending the number field F by the polynomials s_i or the polynomials in the sequence s. Similar as for `NumberField(S)` described above, F may be $\mathbf{Q}$. A tower of fields similar to that of `NumberField` is created and the same restrictions as for that function apply to the polynomials that can be used in the constructor.

Example H34E2

To create the number field $\mathbf{Q}(\alpha)$, where α is a zero of the integer polynomial $x^4 - 420x^2 + 40000$, one may proceed as follows:

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> Degree(K);
> K;
Number Field with defining polynomial x^4 - 420*x^2 + 40000 over the Rational
Field
> y^4 - 420*y^2;
-40000
```

By assigning the generating element to y, we can from here on specify elements in the field as polynomials in y. The elements will always be printed as polynomials in $\mathbf{Q}[y]/f$:

```
> z := y^5/11;
> z;
1/11*(420*y^3 - 40000*y)
```

K can be further extended by the use of either `ext` or `NumberField`.

```
> R<y> := PolynomialRing(K);
> f := y^2 + y + 1;
> L := ext<K | f>;
> L;
Number Field with defining polynomial y^2 + y + 1 over K
```

This is equivalent to

```
> KL := NumberField([x^2 + x + 1, x^4 - 420*x^2 + 40000]);
> KL;
Number Field with defining polynomial $.1^2 + $.1 + 1 over its ground field
```

but different to

```
> LK := NumberField([x^4 - 420*x^2 + 40000, x^2 + x + 1]);
> LK;
Number Field with defining polynomial $.1^4 - 420*$.1^2 + 40000 over its ground
field
```

To illustrate the use of `Global`:

```
> K1 := NumberField(x^3-2 : Global);
> K2 := NumberField(x^3-2 : Global);
> L1 := NumberField(x^3-2);
> L2 := NumberField(x^3-2);
> K1 eq K2;
true
> K1 eq L1;
false
> L1 eq L2;
false;
> K1!K2.1;
K1.1;
>  K2!K1.1;
K1.1
>> L1!L2.1;
     ^
Runtime error in '!': Arguments are not compatible
LHS: FldNum
RHS: FldNumElt
```

A typical application of `DoLinearExtension` is as follows. To construct a Kummer extension of degree p, one has to start with a field containing the p-th roots of unity. In most situation this will be a field extension of degree $p-1$, but what happens if ζ_p is already in the base field?

```
> AdjoinRoot := function(K, p: DoLinearExtension := false)
>   f := CyclotomicPolynomial(p);
>   f := Polynomial(K, f);
>   f := Factorisation(f)[1][1];
>   return ext<K|f : DoLinearExtension := DoLinearExtension>;
> end function;
> K := NumberField(x^2+x+1);
> E1 := AdjoinRoot(K, 3);
> E1;
Number Field with defining polynomial x^2 + x + 1 over the
Rational Field
> E2 := AdjoinRoot(K, 3 : DoLinearExtension);
> E2;
Number Field with defining polynomial ext<K|>.1 - K.1 over
K
```

```
> Norm(E1.1);
1
> Norm(E2.1);
K.1
> Norm($1);
1
```

`RadicalExtension(F, d, a)`

Check	BoolElt	*Default :* `true`

Let F be a number field. Let a be an integral element of F chosen such that a is not an n-th power for any n dividing d. Returns the number field obtained by adjoining the d-th root of a to F.

`SplittingField(F)`

Abs	BoolElt	*Default :* `true`
Opt	BoolElt	*Default :* `true`

Given a number field F, return the splitting field of its defining polynomial. The roots of the defining polynomial in the splitting field are also returned.

If `Abs` is `true`, the resulting field will be an absolute extension, otherwise a tower is returned.

If `Opt` is `true`, an attempt of using `OptimizedRepresentation` is done. If successful, the resulting field will have a much nicer representation. On the other hand, computing the intermediate maximal orders can be extremely time consuming.

`SplittingField(f)`

Given an irreducible polynomial f over $\mathbf{Z}$, return its splitting field.

`SplittingField(L)`

Abs	BoolElt	*Default :* `false`
Opt	BoolElt	*Default :* `false`

Given a sequence L of polynomials over a number field or the rational numbers, compute a common splitting field, ie. a field K such that every polynomial in L splits into linear factors over K. The roots of the polynomials are returned as the second return value.

If the optional parameter `Abs` is `true`, then a primitive element for the splitting field is computed and the field returned will be generated by this primitive element over $\mathbf{Q}$. If in addition `Opt` is also `true`, then an optimized representation of K is computed as well.

`sub< F | e_1, ..., e_n >`

Given a number field F with coefficient field G and n elements $e_i \in F$, return the number field $H = G(e_1, \ldots, e_n)$ generated by the e_i (over G), as well as the embedding homomorphism from H to F.

`MergeFields(F, L)`

`CompositeFields(F, L)`

Let F and L be absolute number fields. Returns a sequence of fields $[M_1, \ldots, M_r]$ such that each field M_i contains both a root of the generating polynomial of F and a root of the generating polynomial of L.

In detail: Suppose that F is the smaller field (wrt. the degree). As a first step we factorise the defining polynomial of L over F. For each factor obtained, an extension of F is constructed and then transformed into an absolute extension. The sequence of extension fields is returned to the user.

`Compositum(K, L)`

For absolute number fields K and L, at least one of which must be normal, find a smallest common over field. Note that in contrast to `CompositeFields` above the result here is essentially unique since one field was normal.

`quo< FldNum : R | f >`

`Check` BoolElt *Default :* `true`

Given a ring of polynomials R in one variable over a number field K, create the number field $K(\alpha)$ obtained by adjoining a root α of f to K. Here the coefficient ring K of R is allowed to be the rational field $\mathbf{Q}$. The polynomial f is allowed to have coefficients in K, but if coefficients occur in f which require denominator greater than 1 when expressed on the basis of K, the polynomial will be replaced by an equivalent one requiring no such denominators: $\tilde{f}(x) = df(x)$, where d is a common denominator. The parameter `Check` determines whether the polynomial is checked for irreducibility.

The angle bracket notation may be used to assign the root α to an identifier: `K<y> := quo< FldNum : R | f >`.

If the category `FldNum` is not specified, `quo< R | f >` creates the quotient ring R/f as a generic ring (not as a number field), in which only elementary arithmetic is possible.

Example H34E3

To illustrate the use of `CompositeFields` we will use this function to compute the normal closure of $\mathbf{Q}(\alpha)$ where α is a zero of the integer polynomial $x^3 - 2$:

```
> K := RadicalExtension(Rationals(), 3, 2);
> l := CompositeFields(K, K);
> l;
[
    Number Field with defining polynomial $.1^3 - 2 over the Rational
    Field,
    Number Field with defining polynomial $.1^6 + 108 over the Rational
    Field
]
```

The second element of l corresponds to the smallest field L_2 containing two distinct roots of x^3-2. Since the degree of K is 3, L_2 is the splitting field of f and therefore the normal closure of K.

`OptimizedRepresentation(F)`

`OptimisedRepresentation(F)`

Given a number field F with ground field $\mathbf{Q}$, this function will attempt to find an isomorphic field L with a better defining polynomial than the one used to define F. If such a polynomial is found then L is returned; otherwise F will be returned. For more details, please refer to `OptimizedRepresentation`.

Example H34E4

Some results of `OptimizedRepresentation` are shown.

```
> R<x> := PolynomialRing(Rationals());
> K := NumberField(x^4-420*x^2+40000);
> L := OptimizedRepresentation(K);
> L ne K;
true
> L;
Number Field with defining polynomial x^4 - 4*x^3 -
    17*x^2 + 42*x + 59 over the Rational Field
> L eq OptimizedRepresentation(L);
```

34.2.2 Maximal Orders

The maximal order $\mathcal{O}_K$ is the ring of integers of an algebraic field consisting of all integral elements of the field; that is, elements which are roots of monic integer polynomials. It may also be called the number ring of a number field. It is arguably the single most important invariant of a number field, in fact in number theory when one talks about units, ideals, etc. of number fields, it is typically implied that the maximal order is the underlying ring.

Maximal orders and orders in general are explained in detail in the chapter 37, here we only give a very brief overview.

There are a number of algorithms which MAGMA uses whilst computing maximal orders. The main ones are the Round–2 and the Round–4 methods ([Coh93, Bai96, Poh93, PZ89] for absolute extensions and [Coh00, Fri97, Pau01b] for relative extensions).

```
MaximalOrder(F)
```
```
IntegerRing(F)
```
```
Integers(F)
```
```
RingOfIntegers(F)
```

`Al`	MonStgElt	*Default : "Auto"*
Verbose	`MaximalOrder`	*Maximum :* 5

Create the ring of integers of the algebraic number field F. An integral basis for F can be found as the basis of the maximal order.

For information on the parameters, see Section 37.2.3.

34.2.3 Creation of Elements

Since number fields are though of as quotients of (multivariate) polynomial rings, elements in those fields are represented as (multivariate) polynomials in the generator(s) of the field.

```
F ! a
```
```
elt< F | a >
```

Coerce a into the number field F. Here a may be an integer or a rational field element, or an element from a subfield of F, or from an order in such or any other field related to F through chains of subfields, optimised representation, absolute fields, etc.

`F ! [`a_0`, `a_1`, ..., `a_{m-1}`]`

`elt< F | [ `a_0`, `a_1`, ..., `a_{m-1}` ] >`

`elt< F | `a_0`, `a_1`, ..., `a_{m-1}` >`

Given the number field, F of degree m over its ground field G and a sequence $[a_0, \ldots, a_{m-1}]$ of elements of G, construct the element $a_0\alpha_0 + a_1\alpha_1 + \cdots a_{m-1}\alpha_{m-1}$ of F where the α_i are the basis elements of F. In case F was generated by a root of a single polynomial, we will always have $\alpha_i = \mathbf{F.1}^i$. If F was defined using multiple polynomials and the `Abs` parameter, the basis will consist of products of powers of the generators.

```
Random(F, m)
```

A random element of the number field F. The maximal size of the coefficients is determined by the integer m.

Example H34E5

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^4-420*x^2+40000);
> y^6;
136400*y^2 - 16800000
```

```
> K![-16800000, 0, 136400, 0];
136400*y^2 - 16800000
> K := NumberField([x^3-2, x^2-5]:Abs);
> Basis(K);
[
    1,
    K.1,
    K.1^2,
    K.2,
    K.1*K.2,
    K.1^2*K.2
]
> K![1,2,3,4,5,6];
6*K.1^2*K.2 + 3*K.1^2 + 5*K.1*K.2 + 2*K.1 + 4*K.2 + 1
```

`One(K)` `Identity(K)`

`Zero(K)` `Representative(K)`

34.2.4 Creation of Homomorphisms

To specify homomorphisms from number fields, it is necessary to specify the image of the generating elements, and possible to specify a map on the coefficient field.

`hom< F -> R | r >`

`hom< F -> R | h, r >`

`hom< F -> R | r >`

`hom< F -> R | h, r >`

Given an algebraic number field F, defined as an extension of the coefficient field G, as well as some ring R, build the homomorphism ϕ obtained by sending the defining primitive element α of F to the element $r \in R$.

In case the field F was defined using multiple polynomials, instead of an image for the primitive element, one has to give images for each of the generators.

It is possible (if $G = \mathbf{Q}$) and sometimes necessary (if $G \neq \mathbf{Q}$) to specify a homomorphism ϕ on F by specifying its action on G by providing a homomorphism h with G as its domain and R its codomain together with the image of α. If R does not cover G then the homomorphism h from G into R is necessary to ensure that the ground field can be mapped into R.

Example H34E6

We show a way to embed the field $\mathbf{Q}(\sqrt{2})$ in $\mathbf{Q}(\sqrt{2}+\sqrt{3})$. The application of the homomorphism suggests how the image could have been chosen.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^2-2);
> KL<w> := NumberField(x^4-10*x^2+1);
> H := hom< K -> KL | (9*w-w^3)/2 >;
> H(y);
1/2*(-w^3 + 9*w)
> H(y)^2;
2
```

34.3 Structure Operations

In the lists below K always denotes a number field.

34.3.1 General Functions

Number fields form the Magma category `FldNum`. The notional power structures exist as parents of algebraic fields with no operations are allowed.

`Category(K)` `Type(K)` `ExtendedType(K)` `Parent(K)`

`AssignNames(~K, s)`

Procedure to change the names of the generating elements in the number field K to the contents of the sequence of strings s.

The i-th sequence element will be the name used for the generator of the $(i-1)$-st subfield down from K as determined by the creation of K, the first element being used as the name for the generator of K. In the case where K is defined by more than one polynomial as an absolute extension, the ith sequence element will be the name used for the root of the ith polynomial used in the creation of K.

This procedure only changes the names used in printing the elements of K. It does *not* assign to any identifiers the value of a generator in K; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies K, it is necessary to have a reference `~K` to K in the call to this function.

`Name(K, i)`

`K . i`

Given a number field K, return the element which has the i-th name attached to it, that is, the generator of the $(i-1)$-st subfield down from K as determined by the creation of K. Here i must be in the range $1 \leq i \leq m$, where m is the number of polynomials used in creating K. If K was created using multiple polynomials as an absolute extension, `K.i` will be a root of the ith polynomial used in creating K.

34.3.2 Related Structures

Each number field has other structures related to it in various ways.

`GroundField(F)`

`BaseField(F)`

`CoefficientField(F)`

`CoefficientRing(F)`

Given a number field F, return the number field over which F was defined. For an absolute number field F, the function returns the rational field $\mathbf{Q}$.

`AbsoluteField(F)`

Given a number field F, this returns an isomorphic number field L defined as an absolute extension (i.e. over $\mathbf{Q}$). (For algorithm, see [Tra76])

`SimpleExtension(F)`

Given a number field F or an order O, this returns an isomorphic field L defined as an absolute simple extension. (For algorithm, see [Tra76])

`RelativeField(F, L)`

Given number fields L and F such that Magma knows that F is a subfield of L, return an isomorphic number field M defined as an extension over F.

Example H34E7

It is often desirable to build up a number field by adjoining several algebraic numbers to $\mathbf{Q}$. The following function returns a number field that is the composite field of two given number fields K and L, provided that $K \cap L = \mathbf{Q}$; if K and L have a common subfield larger than $\mathbf{Q}$ the function returns a field with the property that it contains a subfield isomorphic to K as well as a subfield isomorphic to L.

```
> R<x> := PolynomialRing(Integers());
> Composite := function( K, L )
>     T<y> := PolynomialRing( K );
>     f := T!DefiningPolynomial( L );
>     ff := Factorization(f);
>     LKM := NumberField(ff[1][1]);
>     return AbsoluteField(LKM);
> end function;
```

To create, for example, the field $\mathbf{Q}(\sqrt{2}, \sqrt{3}, \sqrt{5})$, the above function should be applied twice:

```
> K := NumberField(x^2-3);
> L := NumberField(x^2-2);
> M := NumberField(x^2-5);
> KL := Composite(K, L);
> S<s> := PolynomialRing(BaseField(KL));
> KLM<w> := Composite(KL, M);
> KLM;
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

Note, that the same field may be constructed with just one call to `NumberField` followed by `AbsoluteField`:

```
> KLM2 := AbsoluteField(NumberField([x^2-3, x^2-2, x^2-5]));
> KLM2;
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

or by

```
> AbsoluteField(ext<Rationals() | [x^2-3, x^2-2, x^2-5]>);
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

In general, however, the resulting polynomials of KLM and $KLM2$ will differ. To see the difference between `SimpleExtension` and `AbsoluteField`, we will create `KLM2` again:

```
> KLM3 := NumberField([x^2-3, x^2-2, x^2-5]: Abs);
> AbsoluteField(KLM3);
Number Field with defining polynomials [ x^2 - 3, x^2 - 2,
    x^2 - 5] over the Rational Field
> SimpleExtension(KLM3);
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

`PrimeRing(F)` `PrimeField(F)`

`Centre(F)`

`Embed(F, L, a)`

Install the embedding of a simple number field F in L where the image of the primitive element of F is the element a of L. This embedding will be used in coercing from F into L.

`Embed(F, L, a)`

Install the embedding of the non-simple number field F in L where the image of the generating elements of F are in the sequence a of elements of L. This embedding will be used in coercing from F into L.

`EmbeddingMap(F, L)`

Returns the embedding map of the number field F in L if an embedding is known.

Example H34E8

Magma does not recognize two independently created number fields as equal since more than one embedding of a field in a larger field may be possible. To coerce between them, it is convenient to be able to embed them in each other.

```
> k := NumberField(x^2-2);
> l := NumberField(x^2-2);
>  l!k.1;
>> l!k.1;
    ^
Runtime error in '!': Arguments are not compatible
LHS: FldNum
RHS: FldNumElt
> l eq k;
false
> Embed(k, l, l.1);
> l!k.1;
l.1
> Embed(l, k, k.1);
> k!l.1;
k.1
```

`Embed` is useful in specifying the embedding of a field in a larger field.

```
> l<a> := NumberField(x^3-2);
> L<b> := NumberField(x^6+108);
> Root(L!2, 3);
1/18*b^4
> Embed(l, L, $1);
> L!l.1;
1/18*b^4
```

Another embedding would be

```
> Roots(PolynomialRing(L)!DefiningPolynomial(l));
[
    <1/36*(-b^4 - 18*b), 1>,
    <1/36*(-b^4 + 18*b), 1>,
    <1/18*b^4, 1>
]
> Embed(l, L, $1[1][1]);
> L!l.1;
1/36*(-b^4 - 18*b)
```

`MinkowskiSpace(F)`

The Minkowski vector space V of the absolute number field F as a real vector space, with inner product given by the T_2-norm (`Length`) on F, and by the embedding $F \to V$.

```
Completion(K, P)
```

```
comp< K|P >
```

Precision RngIntElt *Default :* 20

For an absolute extension K of $\mathbf{Q}$, compute the completion at a prime ideal P which must be either a prime ideal of the maximal order or unramified. The result will be a local field or ring with relative precision `Precision`.

The returned map is the canonical injection into the completion. It allows pointwise inverse operations.

```
Completion(K, P)
```

Precision RngIntElt *Default :* 20

For an absolute extension K over $\mathbf{Q}$ and a (finite) place P, compute the completion at P. The precision and the map are as described for `Completion`.

34.3.3 Representing Fields as Vector Spaces

It is possible to express a number field as a vector space of any subfield using the intrinsics below. Such a construction also allows one to find properties of elements over these subfields.

```
Algebra(K, J)
```

```
Algebra(K, J, S)
```

Returns the associative structure constant algebra which is isomorphic to the number field K as an algebra over J. Also returns the isomorphism from K to the algebra mapping w^i to the $i+1$st unit vector of the algebra where w is a primitive element of K.

If a sequence S is given it is taken to be a basis of K over J and the isomorphism will map the ith element of S to the ith unit vector of the algebra.

```
VectorSpace(K, J)
```

```
KSpace(K, J)
```

```
VectorSpace(K, J, S)
```

```
KSpace(K, J, S)
```

The vector space isomorphic to the number field K as a vector space over J and the isomorphism from K to the vector space. The isomorphism maps w^i to the $i+1$st unit vector of the vector space where w is a primitive element of K.

If S is given, the isomorphism will map the ith element of S to the ith unit vector of the vector space.

Example H34E9

We use the `Algebra` of a relative number field to obtain the minimal polynomial of an element over a subfield which is not in its coefficient field tower.

```
> K := NumberField([x^2 - 2, x^2 - 3, x^2 - 7]);
> J := AbsoluteField(NumberField([x^2 - 2, x^2 - 7]));
> A, m := Algebra(K, J);
> A;
Associative Algebra of dimension 2 with base ring J
> m;
Mapping from: RngOrd: K to AlgAss: A
> m(K.1);
(1/10*(J.1^3 - 13*J.1)                          0)
> m(K.1^2);
(2 0)
> m(K.2);
(1/470*(83*J.1^3 + 125*J.1^2 - 1419*J.1 - 1735) 1/940*(-24*J.1^3 - 5*J.1^2 +
    382*J.1 + 295))
> m(K.2^2);
(3 0)
> m(K.3);
(1/10*(-J.1^3 + 23*J.1)                          0)
> m(K.3^2);
(7 0)
> A.1 @@ m;
1
> A.2 @@ m;
(($.1 - 1)*$.1 - $.1 - 1)*K.1 + ($.1 + 1)*$.1 + $.1 + 1
>
> r := 5*K.1 - 8*K.2 + K.3;
> m(r);
(1/235*(-238*J.1^3 - 500*J.1^2 + 4689*J.1 + 6940) 1/235*(48*J.1^3 + 10*J.1^2 -
    764*J.1 - 590))
> MinimalPolynomial($1);
$.1^2 + 1/5*(-4*J.1^3 + 42*J.1)*$.1 + 5*J.1^2 - 180
> Evaluate($1, r);
0
> K:Maximal;
  K
  |
  |
  $1
  |
  |
  $2
  |
  |
  Q
```

```
K  : $.1^2 - 2
$1 : $.1^2 - 3
$2 : x^2 - 7
> Parent($3);
Univariate Polynomial Ring over J
> J;
Number Field with defining polynomial $.1^4 - 18*$.1^2 + 25 over the Rational
Field
```

34.3.4 Invariants

Some information describing a number field can be retrieved.

`Characteristic(F)`

`Degree(F)`

Given a number field F, return the degree $[F : G]$ of F over its ground field G.

`AbsoluteDegree(F)`

Given a number field F, return the absolute degree of F over $\mathbf{Q}$.

`Discriminant(F)`

Given an extension F of $\mathbf{Q}$, return the discriminant of F. This discriminant is defined to be the discriminant of the defining polynomial, **not** as the discriminant of the maximal order.

The discriminant in a relative extension F is the ideal in the base ring generated by the discriminant of the defining polynomial.

`AbsoluteDiscriminant(K)`

Given a number field K, return the absolute value of the discriminant of K regarded as an extension of $\mathbf{Q}$.

`Regulator(K)`

Given a number field K, return the regulator of K as a real number. Note that this will trigger the computation of the maximal order and its unit group if they are not known yet. This only works in an absolute extension.

`RegulatorLowerBound(K)`

Given a number field K, return a lower bound on the regulator of O or K. This only works in an absolute extension.

`Signature(F)`

Given an absolute number field F, returns two integers, one being the number of real embeddings, the other the number of pairs of complex embeddings of F.

`UnitRank(K)`

The unit rank of the number field K (one less than the number of real embeddings plus number of pairs of complex embeddings).

`DefiningPolynomial(F)`

Given a number field F, the polynomial defining F as an extension of its ground field G is returned.

For non simple extensions, this will return a list of polynomials.

`Zeroes(F, n)`

Given an absolute number field F, and an integer n, return the zeroes of the defining polynomial of F with a precision of exactly n decimal digits. The function returns a sequence of length the degree of F; all of the real zeroes appear before the complex zeroes.

Example H34E10

The information provided by `Zeros` and `DefiningPolynomial` is illustrated below.

```
> L := NumberField(x^6+108);
> DefiningPolynomial(L);
x^6 + 108
> Zeros(L, 30);
[ 1.88988157484230974715081591089999999994 +
1.09112363597172140356007261419999999977*i,
1.88988157484230974715081591089999999994 -
1.09112363597172140356007261419999999977*i,  0.E-29 +
2.18224727194344280712014522839999999955*i,  0.E-29 -
2.18224727194344280712014522839999999955*i,
-1.88988157484230974715081591089999999994 +
1.09112363597172140356007261419999999977*i,
-1.88988157484230974715081591089999999994 -
1.09112363597172140356007261419999999977*i ]
> l := NumberField(x^3 - 2);
> DefiningPolynomial(l);
x^3 - 2
> Zeros(l, 30);
[ 1.25992104989487316476721060729999999994,
-0.62996052494743658238360530363910999999 +
1.09112363597172140356007261419999999977*i,
-0.62996052494743658238360530363910999999 -
1.09112363597172140356007261419999999977*i ]
```

34.3.5 Basis Representation

The basis of a number field can be expressed using elements from any compatible ring.

```
Basis(F)
Basis(F, R)
```

Return the current basis for the number field F over its ground ring as a sequence of elements of F or as a sequence of elements of R.

```
IntegralBasis(F)
IntegralBasis(F, R)
```

An integral basis for the algebraic number field F is returned as a sequence of elements of F or R if given. This is the same as the basis for the maximal order. Note that the maximal order will be determined (and stored) if necessary.

Example H34E11

The following illustrates how a basis can look different when expressed in a different ring.

```
> f := x^5 + 5*x^4 - 75*x^3 + 250*x^2 + 65625;
> N := NumberField(f);
> N;
Number Field with defining polynomial x^5 + 5*x^4 - 75*x^3 + 250*x^2 + 65625
over the Rational Field
> Basis(N);
[
    1,
    N.1,
    N.1^2,
    N.1^3,
    N.1^4
]
> IntegralBasis(N);
[
    1,
    1/5*N.1,
    1/25*N.1^2,
    1/125*N.1^3,
    1/625*N.1^4
]
> IntegralBasis(N, MaximalOrder(N));
[
    [1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 1]
]
```

`AbsoluteBasis(K)`

Returns an absolute basis for the number field K, i.e. a basis for K as a **Q** vector space. The basis will consist of the products of the basis elements of the intermediate fields. The expansion is done depth-first.

Example H34E12

We continue our example of a field of degree 4.
The functions `Basis` and `IntegralBasis` both return a sequence of elements, that can be accessed using the operators for enumerated sequences. Note that if, as in our example, O is the maximal order of K, both functions produce the same output:

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> O := MaximalOrder(K);
> I := IntegralBasis(K);
> B := Basis(O);
> I, B;
[
    1,
    1/2*y,
    1/40*(y^2 + 10*y),
    1/800*(y^3 + 180*y + 400)
]
[
    O.1,
    O.2,
    O.3,
    O.4
]
> Basis(O, K);
[
    1,
    1/2*y,
    1/40*(y^2 + 10*y),
    1/800*(y^3 + 180*y + 400)
]
```

34.3.6 Ring Predicates

Number fields can be tested for having several properties that may hold for general rings.

`F eq L`

Returns `true` if and only if the number fields F and L are indentical.

No two number fields which have been created independently of each other will be considered equal since it is possible that they can be embedded into a larger field in more than one way.

`IsCommutative(R)` `IsUnitary(R)` `IsFinite(R)`

`IsOrdered(R)` `IsField(R)`

`IsNumberField(R)` `IsAlgebraicField(R)`

`IsEuclideanDomain(F)`

This is not a check for euclidean number fields. This function will always return `true`, as all number fields are euclidean domains.

`IsSimple(F)`

Checks if the number field F is defined as a simple extension over the base ring.

`IsPID(F)` `IsUFD(F)`

`IsPrincipalIdealRing(F)`

Always `true` for number fields.

`IsDomain(R)`

`F ne L` `K subset L`

`HasComplexConjugate(K)`

This function returns `true` if there is an automorphism in the number field K that acts like complex conjugation.

`ComplexConjugate(x)`

For an element x of a number field K where `HasComplexConjugate` returns `true` (in particular this includes totally real fields, cyclotomic and quadratic fields and CM-extensions), the conjugate of x is returned.

34.3.7 Field Predicates

Here all the predicates that are specific to number fields are listed.

`IsIsomorphic(F, L)`

Given two number fields F and L, this returns `true` as well as an isomorphism $F \to L$, if F and L are isomorphic, and it returns `false` otherwise.

`IsSubfield(F, L)`

Given two number fields F and L, this returns `true` as well as an embedding $F \hookrightarrow L$, if F is a subfield of L, and it returns `false` otherwise.

`IsNormal(F)`

Returns `true` if and only if the number field F is a normal extension. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois group computation.

`IsAbelian(F)`

Returns `true` if and only if the number field F is a normal extension with abelian Galois group. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois Group computation.

`IsCyclic(F)`

Returns `true` if and only if the number field F is a normal extension with cyclic Galois group. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois and automorphism group.

`IsAbsoluteField(K)`

Returns `true` iff the number field K is a constructed as an absolute extension of $\mathbf{Q}$.

34.4 Element Operations

34.4.1 Parent and Category

`Parent(a)` `Category(a)` `Type(a)` `ExtendedType(a)`

34.4.2 Arithmetic

The table below lists the generic arithmetic functions on number field elements. Note that automatic coercion ensures that the binary operations +, -, *, and / may be applied to an element of a number field and an element of one of its orders; the result will be a number field element.

`+ a` `- a`

`a + b` `a - b` `a * b` `a / b` `a ^ k`

`Sqrt(a)`

Returns the square root of the number field element a if it exists in the field containing a.

`Root(a, n)`

Returns the n-th root of the number field element a if it exists in the field containing a.

`IsSquare(a)`

Return `true` if the number field element a is a kth power, (respectively square) and the root if so.

`Denominator(a)`

Returns the denominator of the number field element a, that is the least common multiple of the denominators of the coefficients of a.

`Numerator(a)`

Returns the numerator of the number field element a, that is the element multiplied by its denominator.

`Qround(E, M)`

`ContFrac` BoolElt *Default :* `true`

Finds an approximation of the number field element E where the denominator is bounded by the integer M. If `ContFrac` is `true`, the approximation is computed by applying the continued fraction algorithm to the coefficients of E viewed over Q.

34.4.3 Equality and Membership

Elements may also be tested for whether they lie in an ideal of an order. See Section 37.9.5.

`a eq b` `a ne b`

`a in F`

34.4.4 Predicates on Elements

In addition to the generic predicates `IsMinusOne`, `IsZero` and `IsOne`, the predicates `IsIntegral` and `IsPrimitive` are defined on elements of number fields.

`IsIntegral(a)`

Returns `true` if the element a of a number field F is contained in the ring of integers of F, `false` otherwise. We use the minimal polynomial to determine the answer, which means that the calculation of the maximal order is *not* triggered if it is not known yet. A denominator d such that $d * a$ is integral is also returned on request.

`IsPrimitive(a)`

Returns `true` if the element a of the number field F generates F over its coefficient field.

`IsTotallyPositive(a)`

Returnes `true` iff all real embeddings of the number field element a are positive. For elements in absolute fields this is equivalent to all real conjugates being positive.

`IsZero(a)` `IsOne(a)`

`IsMinusOne(a)` `IsUnit(a)`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsZeroDivisor(a)` `IsRegular(a)`

`IsIrreducible(a)` `IsPrime(a)`

34.4.5 Finding Special Elements

Generators of fields can be retrieved.

`K . 1`

Return the image α of x in $G[x]/f$ where f is the first defining polynomial of the number field K and G is the base field of K.

In case of simple extensions this will be a primitive element.

`PrimitiveElement(K)`

Returns a primitive element for the simple number field K, that is an element whose minimal polynomial has the same degree as the field. For a simple number field K this is $K.1$, while for non-simple fields a random element with this property is returned.

`Generators(K)`

The set of generators of the number field K over its coefficient field, that is a set containing a root of each defining polynomial is returned.

```
GeneratorsOverBaseRing(K)
```

A set of generators of the number field K over $\mathbf{Q}$.

```
GeneratorsSequence(K)
```

The sequence of generators of the number field K over its coefficient field, that is a sequence containing a root of each defining polynomial is returned.

```
GeneratorsSequenceOverBaseRing(K)
```

A sequence of generators of the number field K over $\mathbf{Q}$.

```
Generators(K, k)
```

A sequence of generators of the number field K over k is returned. That is a sequence containing a root of each defining polynomial for K and its subfield down to the level of k is returned.

34.4.6 Real and Complex Valued Functions

The functions here return (sequences of) real or complex numbers. The precision of these numbers is governed by the appropriate or field's internal precision. See Section 37.3 for more information.

```
AbsoluteValues(a)
```

Return a sequence of length $r_1 + r_2$ of the real absolute values of the conjugates of the number field element a. The first r_1 values are the absolute values of the real embeddings of the element, the next r_2 are the lengths of the complex embeddings with their weight factors. That is, if the real conjugates of a are w_i, for $1 \leq i \leq r_1$, and the complex conjugates of a are $x_i \pm iy_i$ (for $1 \leq i \leq r_2$), then `AbsoluteValues` returns $[|w_1|, \ldots, |w_{r_1}|, \sqrt{\frac{x_{r_1+1}^2 + y_{r_1+1}^2}{2}}, \ldots \sqrt{\frac{x_{r_1+r_2}^2 + y_{r_1+r_2}^2}{2}}]$.

```
AbsoluteLogarithmicHeight(a)
```

Let P be the minimal polynomial of the number field element a over $\mathbf{Z}$, with leading coefficient a_0 and roots $\alpha_1, \ldots, \alpha_n$. Then the absolute logarithmic height is defined to be

$$h(\alpha) = \frac{1}{n} \log(a_0 \prod_{j=1}^{n} \max(1, |\alpha_j|)).$$

```
Conjugates(a)
```

The real and complex conjugates of the given algebraic number a, as a sequence of n complex numbers. The r_1 real conjugates appear first, and are followed by r_2 pairs of complex conjugates. The field should be an absolute extension. The ordering of the conjugates is consistent for elements of the same field (or even for elements of different fields that have the same defining polynomial).

`Conjugate(a, k)`

Equivalent to `Conjugates(a)[k]`.

`Conjugate(a, l)`

Let $l := [l_1, \dots, l_n]$ be a sequence of positive integers and assume that the number field K, the parent of a is given as a tower with n steps, $\mathbf{Q} \subseteq K_1 \subseteq \dots \subseteq K_n = K$. This function computes the image of a in $\mathbf{C}$ or $\mathbf{R}$ under the embedding determined by l, that is under embedding obtained by extending the l_1 embedding of K_1 to K_2, then extending the l_2nd of those embeddings to K_3 and so on.

`Length(a)`

Return the T_2-norm of the number field element a, which is a real number. This equals the sum of the (complex) norms of the conjugates of a.

`Logs(a)`

Return the sequence of length $r_1 + r_2$ of logarithms of the absolute values of the conjugates of a number field element $a \neq 0$.

`CoefficientHeight(E)`

Computes the coefficient height of the number field element E, that is for an element of an absolute field it returns the maximum of the denominator and the largest coefficient wrt. to the basis of the parent. For elements in relative extensions, it returns the maximal coefficient height of all the coefficients wrt. the basis of the parent.

This function indicates in some way the difficulty of operations involving this element.

`CoefficientLength(E)`

Computes the coefficient length of the number field element E, that is for an element of an absolute field it returns the sum of the denominator and the absolute values of all coefficients wrt. to the basis of the parent. For elements in relative extensions, it returns the sum of the coefficient length of all the coefficients wrt. the basis of the parent.

This function gives an indication on the amount of memory occupied by this element.

34.4.7 Norm, Trace, and Minimal Polynomial

The norm, trace and minimal polynomial of number field elements can be calculated both with respect to the coefficient ring and to **Z** or **Q**.

```
Norm(a)
```
```
Norm(a, R)
```

The relative norm $\mathrm{N_{L/F}(a)}$ over F of the element a of the number field L where F is the field over which L is defined as an extension. If R is given the norm is calculated over R. In this case, R must occur as a coefficient ring somewhere in the tower under L.

```
AbsoluteNorm(a)
```
```
NormAbs(a)
```

The absolute norm $\mathrm{N_{L/\mathbf{Q}}(a)}$ over **Q** of the element a of the number field L.

```
Trace(a)
```
```
Trace(a, R)
```

The relative trace $\mathrm{Tr}_{L/F}(a)$ over F of the element a of the number field L where F is the field over which L is defined as an extension. If R is given the trace is computed over R. In this case, R must occur as a coefficient ring somewhere in the tower under L.

```
AbsoluteTrace(a)
```
```
TraceAbs(a)
```

The absolute trace $\mathrm{Tr}_{L/\mathbf{Q}}(a)$ over **Q** of the element a of the number field L.

```
CharacteristicPolynomial(a)
```
```
CharacteristicPolynomial(a, R)
```

Given an element a from a number field L, returns the characteristic polynomial of the element over R if given or the subfield F otherwise where F is the field over which L is defined as an extension.

```
AbsoluteCharacteristicPolynomial(a)
```

Given an element a from a number field, this function returns the characteristic polynomial of a over **Q**.

```
MinimalPolynomial(a)
```
```
MinimalPolynomial(a, R)
```

Given an element a from a number field L, returns the minimal polynomial of the element over R if given otherwise the subfield F where F is the field over which L is defined as an extension.

```
AbsoluteMinimalPolynomial(a)
```

Given an element a from a number field, this function returns the minimal polynomial of the element as a polynomial over $\mathbf{Q}$.

```
RepresentationMatrix(a)
RepresentationMatrix(a, R)
```

Return the representation matrix of the number field element a, that is, the matrix which represents the linear map wrt to the field basis, given by multiplication by a. The ith row of the representation matrix gives the coefficients of aw_i with respect to the basis $w_1, \ldots, w_n$.

If R is given the matrix is over R and with respect to the basis of the order or field over R.

```
AbsoluteRepresentationMatrix(a)
```

Return the representation matrix of the number field element a relative to the $\mathbf{Q}$-basis of the field constructed using products of the basis elements, where a is an element of the relative number field L.

Let $L_i := \sum L_{i-1}\omega_{i,j}$, $L := L_n$ and $L_0 := \mathbf{Q}$. Then the representation matrix is computed with respect to the $\mathbf{Q}$-basis $(\prod_j \omega_{i_j,j})_{i\in I}$ consisting of products of basis elements of the different levels.

Example H34E13

We create the norm, trace, minimal polynomial and representation matrix of the element $\alpha/2$ in the quartic field $\mathbf{Q}(\alpha)$.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^4-420*x^2+40000);
> z := y/2;
> Norm(z), Trace(z);
2500 0
> MinimalPolynomial(z);
ext<Q|>.1^4 - 105*ext<Q|>.1^2 + 2500
> RepresentationMatrix(z);
[     0    1/2      0      0]
[     0      0    1/2      0]
[     0      0      0    1/2]
[-20000      0    210      0]
```

The awkwardness of the printing of the minimal polynomial above can be overcome by providing a parent for the polynomial, keeping in mind that it is a univariate polynomial over the rationals:

```
> P<t> := PolynomialRing(RationalField());
> MinimalPolynomial(z);
t^4 - 105*t^2 + 2500
```

34.4.8 Other Functions

Elements can be represented by sequences and have a product representation.

```
Eltseq(a)
```

For an element a of a number field F, a sequence of coefficients of length degree of F with respect to the basis is returned.

```
Eltseq(E, k)
```

For an algebraic number $E \in K$ and a ring k which occurs somewhere in the defining tower for K, return the list of coefficients of E over k, that is, apply `Eltseq` to E and to its coefficients until the list is over k.

```
Flat(e)
```

The coefficients of the number field element e wrt. to the canonical Q basis for its field. This is performed by iterating `Eltseq` until the coefficients are rational numbers. The coefficients obtained match the coefficients wrt. to `AbsoluteBasis`.

```
a[i]
```

The coefficient of the ith basis element in the number field element a.

```
ProductRepresentation(a)
```

Return sequences P and E such that the product of elements in P to the corresponding exponents in E is the algebraic number a.

```
ProductRepresentation(P, E)
PowerProduct(P, E)
```

Return the number field element a of the universe of the sequence P such that a is the product of elements of P to the corresponding exponents in the sequence E.

34.5 Class and Unit Groups

This section briefly describes the functions available to compute in and with the ideal and unit group of the number field, that is with the corresponding group in the maximal order. For more details and an idea of the algorithms involved, see Sections 37.6 and 37.7.

All functions mentioned in this section support the verbose flag `ClassGroup` up to a maximum value of 5.

Ideals in MAGMA are discussed in Section 37.9.

`ClassGroup(K: parameters)`

`Bound`	RngIntElt	*Default :* MinkowskiBound
`Proof`	MonStgElt	*Default :* "*Full*"
`Enum`	BoolElt	*Default :* `true`
`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`ClassGroup`	*Maximum :* 5

The group of ideal classes for the ring of integers O of the number field K is returned as an abelian group, together with a map from this abstract group to O. The map admits inverses and can therefore be used to compute "discrete logarithms" for the class group.

With the default values for the optional parameters the Minkowski bound is used and the last step of the algorithm verifies correctness, hence a fully proven result is returned.

If `Bound` is set to some positive integer M, M is used instead of the Minkowski bound. The validity of the result still depends on the `"Proof"` parameter.

If `Proof := "GRH"`, everything remains as in the default case except that a bound based on the GRH is used to replace the Minkowski bound. This bound may be enlarged setting the "Bound" parameter accordingly. The result will hence be correct under the GRH.

If `Proof := "Bound"`, the computation stops if an independent set of relations between the prime ideals below the chosen bound is found. The relations may not be maximal.

If `Proof := "Subgroup"`, a maximal subset of the relations is constructed. In terms of the result, this means that the group returned will be a subgroup of the class group (i.e. the list of prime ideals considered may be to small).

If `Proof := "Full"` (the default) a guaranteed result is computed. This is equivalent to `Bound := MinkowskiBound(K)` and `Proof := "Subgroup"`.

If only `Bound` is given, the `Proof` defaults to `"Subgroup"`.

Finally, giving `Proof := "Current"` is the same as repeating the last call to `ClassGroup()`, but without the need to explicitly restate the value of `Proof` or `Bound`. If there was no prior call to `ClassGroup`, a fully proven computation will be carried out.

If `Enum := false`, then instead of enumerating short elements to get relations, Magma will use random linear combinations of a reduced basis instead. For "small" fields this will typically slow down the computations, but for large fields it is sometimes not possible to find any point using enumeration so that this is necessary for "large" fields. Unfortunately, there is no known criterion to decide beforehand if a field is "large" or "small".

If `Al` is set to `"Sieve"` (regardless of the size of the discriminant) or the discriminant of K is greater than 10^{30} then the sieving method described in [Bia] (or [Jac99] for quadratic fields) will be used. If `Al` is set to `"NoSieve"` then the sieving method will not be used regardless of the size of the discriminant.

```
ConditionalClassGroup(K)
```

The class group of the number field K assuming the generalized Riemann hypothesis.

```
ClassNumber(K: parameters)
```

`Bound`	RngIntElt	*Default :* MinkowskiBound
`Proof`	MonStgElt	*Default :* "*Full*"
`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`ClassGroup`	*Maximum :* 5

Return the class number of the ring of integers O of a number field K. The options for the parameters are the same as for `ClassGroup`.

```
BachBound(K)
```

An integral upper bound for norms of generators of the ideal class group for the number field K assuming the generalized Riemann hypothesis.

```
MinkowskiBound(K)
```

An unconditional integral upper bound for norms of the generators of the ideal class group for the number field K.

```
UnitGroup(K)
MultiplicativeGroup(K)
```

`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`UnitGroup`	*Maximum :* 6

Given a number field K, this function returns an (abstract) abelian group U, as well as a bijection m between U and the units of the maximal order. The unit group consists of the torsion subgroup, generated by the image `m(U.1)` and a free part, generated in O by the images `m(U.i)` for $2 \le i \le r_1 + r_2$.

The parameter `Al` can be used to specify an algorithm. It should be one of `"Automatic"`, (default, a choice will be made for the user) `"ClassGroup"`, `"Dirichlet"`, `"Mixed"` (the best known Dirichlet method), `"Relation"` or `"Short"` (which is a variation of `"Mixed"`). In the case of real quadratic fields, a continued fraction algorithm is available, `"ContFrac"`.

```
TorsionUnitGroup(K)
```

The torsion subgroup of the unit group of the number field K, i.e. its maximal order. The torsion subgroup is returned as an abelian group T, together with a map m from the group to the order O. The torsion subgroup will be cyclic, and is generated by `m(T.1)`.

```
UnitRank(K)
```

Return the unit rank of the ring of integers O of a number field K.

Example H34E14

In our field defined by $x^4 - 420 * x^2 + 40000$, we obtain the class and unit groups as follows.

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> C := ClassGroup(K);
> C;
Abelian Group of order 1
> U := UnitGroup(K);
> U;
Abelian Group isomorphic to Z/2 + Z + Z + Z
Defined on 4 generators
Relations:
      2*U.1 = 0
> T := TorsionUnitGroup(K);
> T;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*T.1 = 0
```

34.6 Galois Theory

`GaloisGroup(K)`

`Subfields(K)`

`AutomorphismGroup(K)`

See Sections 38.2, 38.3 and 38.1.

34.7 Solving Norm Equations

Magma can solve norm, Thue, index form and unit equations. In this section, we will only dicuss norm equations, other types of Diophantine equations are discussed in 37.8.

Norm equations in the context of number fields occur in many applications. While Magma contains efficient algorithms to solve norm equations it is important to understand the difference between the various types of norm equations that occur. Given some element θ in a number field k together with a finite extension K/k, there are two different types of norm equations attached to this data:

- Diophantine norm equations, that is norm equations where a solution $x \in K$ is restricted to a particular order (or any additive subgroup), and
- field theoretic norm equations where any element in $x \in K$ with $N(x) = \theta$ is a solution.

While in the first case the number of *different* (up to equivalence) solutions is finite, no such restriction holds in the field case. On the other hand, the field case often allows to prove the existence or non-existence of solutions quickly, while no efficient tests exist for the Diophantine case. So it is not surprising that different methods are applied for the different cases.

```
NormEquation(F, m)
```

`Primes`	eseq of prime ideals	*Default :* []
`Nice`	BoolElt	*Default :* `true`

Given a number field F and an element m of the base field of F, this function returns a boolean indicating whether an element $\alpha \in F$ exists such that $\mathrm{N}_{\mathrm{F/L}}(\alpha)$, the norm of α with respect to the base field L of F equals m, and if so, a sequence of length 1 of solutions α.

The field theoretic norm equations are all solved using S-units. Before discussing some details, we outline the method.

- Determine a set S of prime ideals. We try to obtain a solution as a S-unit for this set S.
- Compute a basis for the S-units
- Compute the action of the norm-map
- Obtain a solution as a preimage.

In general, no effective method is known for the first step. If the field is relative normal however, it is known that is S generates the class group of F and if m is a S-unit, then S is large enough (*suitable* in ([Coh00, 7.5]) [Fie97, Sim02, Gar80]. Thus to find S we have to compute the class group of F. If a (conditional) class group is already known, it is used, otherwise an *unconditional* class group is computed. The initial set S consists of all prime ideals occurring in the decomposition of $m\mathbf{Z}_F$. Note that this step includes the factorisation of m and thus can take a long time is m is large.

Next, we determine a basis for the S-unit group and the action of the norm on it. This give the norm map as a map on the S-unit group as an abstract abelian group.

Finally, the right hand side m is represented as an element of the S-unit group and a solution is then obtained as a preimage under the norm map.

If `Nice` is true, then MAGMA attempts to find a smaller solution by applying a LLL reduction to the original solution.

If `Primes` is give it must contain a list of prime ideals of L. Together with the primes dividing m it is used to form the set S bypassing the computation of an unconditional class group in this step. If L is not normal this can be used to guarantee that S is large enough. Note that the class group computation is still performed when the S-units are computed. Since the correctness of the S-unit group (we need only p-maximality for all primes dividing the (relative) degree of L) can be verified independently of the correctness of the class group, this can be used to derive provable results in cases where the class group cannot be computed unconditionally.

By default, the `MaximalOrder(L)` is used to compute the class group. If the attribute `NeqOrder` is set on L it must contain a maximal order of L. If present, this order will be used for all the subsequent computations.

`NormEquation(m, N)`

`Raw`	BOOLELT	*Default :* `false`
`Primes`	ESEQ OF PRIME IDEALS	*Default :* []

Let N be a map on the multiplicative group of some number field. Formally N may also be defined on the maximal order of the field. This intrinsic tries to find a pre-image for the element m under N.

This function works by realising N as a endomorphism of S-units for a suitable set S.

If N is a relative norm and if L is (absolutely) normal then the set S as computed for the field theoretic norm equation is guaranteed to be large enough to find a solution if it exists. Note: this condition is not checked.

If `Primes` is given it will be supplemented by the primes dividing m and then used as the set S.

If `Raw` is given, the solution is returned as an unevaluated power product. See the example for details.

The main use of this function is for Galois theoretical constructions where the subfields are defined as fields fixed by certain automorphisms. In this situation the norm function can be realised as the product over the fixed group. It is therefore not necessary to compute a (very messy) relative representation of the field.

`SimNEQ(K, e, f)`

`S`	[RNGORDIDL]	*Default :* `false`
`HasSolution`	BOOLELT	*Default :* `false`

For a number field K and subfield elements $e \in k_1$ and $f \in k_2$, try to find a solution to the simultaneous norm equations $N_{K/k_1}(x) = e$ and $N_{K/k_2}(x) = f$. The algorithm proceeds by first guessing a likely set S of prime ideals that will support a solution - it is exists. Initially S will contain all ramified primes in K, the support of e and f and enough primes to generate the class group of K. In case K is normal over Q this set is large enough to support a solution if there is a solution at all. For arbitrary fields that is most likely not the case. However, if `S` is passed in as a parameter then the set used internally will contain at least this set. If `HasSolution` is `true`, MAGMA will add primes to S until a solution has been found. This is useful in situations where for some theoretical reason it is known that there has to be a solution.

Example H34E15

We try to solve $N(x) = 3$ in some relative extension: (Note that since the larger field is a quadratic extension, the second call tells us that there is no integral element with norm 3)

```
> x := PolynomialRing(Integers()).1;
> K := NumberField([x^2-229, x^2-2]);
> NormEquation(K, 3);
true [
  1/3*K.1 - 16/3
]
```

Next we solve the same equation but come from a different angle, we will define the norm map as an element of the group ring and, instead of explicitly computing a relative extension, work instead with the implicit fixed field.

```
> F := AbsoluteField(K);
> t := F!K.2;
> t^2;
2
> A, _, mA := AutomorphismGroup(F);
> S := sub<A | [ x : x in A | mA(x)(t) eq t]>;
> N := map<F -> F | x:-> &* [ mA(y)(x) : y in S]>;
> NormEquation(3, N);
true [
  -5/1*$.1 + 2/3*$.3
]
```

Finally, to show the effect of `Raw`:

```
> f, s, base := NormEquation(3, N:Raw);
> s;
[
  ( 0  1 -1  1 -1  0  2 -1 -1 -1 -1  2  0  0)
]
> z := PowerProduct(base, s[1]);
> z;
-5/1*$.1 + 2/3*$.3
> N(z);
3
```

34.8 Places and Divisors

A place of a number field K, an object of type `PlcNumElt`, is a class of absolute values (valuations) that induce the same topology on the field. By a famous theorem of Ostrowski, places of number fields are either finite, in which case they are in a one-to-one correspondence with the on-zero prime ideals of the maximal order, or infinite. The infinite places are identified with the embedding of K into $\mathbf{R}$ or with pairs of embeddings into $\mathbf{C}$.

The group of divisors is formally the free group generated by the finite places and the $\mathbf{R}$-vectorspace generated by the infinite ones. Divisors are of type `DivNumElt`. Places have formal parent of type `PlcNum`, while divisors belong to `DivNum`.

34.8.1 Creation of Structures

`Places(K)`

`DivisorGroup(K)`

The set of places of the number field K and the group of divisors of K respectively.

34.8.2 Operations on Structures

`d1 eq d2` `p1 eq p2`

`NumberField(P)`

`NumberField(D)`

The number field for which P is the set of places or D is the group of divisors.

34.8.3 Creation of Elements

`Place(I)`

The place corresponding to prime ideal I.

`Decomposition(K, p)`

`Decomposition(K, I)`

A sequence of tuples of places and multiplicities. When a finite prime (integer) p is given, the places and multiplicities correspond to the decomposition of p in the maximal order of K. When the infinite prime is given, a sequence of all infinite places is returned.

`Decomposition(K, p)`

For a number field K and a place p of the coefficient field of K, compute all places (and their multiplicity) that extend p. For finite places this is equivalent to the decomposition of the underlying prime ideal. The sequence returned will contain the places of K extending p and their ramification index.

For an infinite place p, this function will compute all extensions of p in K. In this case, the integer returned in the second component of the tuples will be 1 if p is complex or if p is real and extends to a real place and 2 otherwise.

`Decomposition(m, p)`

`Decomposition(m, p)`

For an extension K/k of number fields (where k can be Q as well), given by the embedding map $m : k \to K$, decompose the place p of k in the larger field. In case $k = Q$, the place is given as either a prime number or zero to indicate the infinite place. The sequence returned contains pairs where the first component is a place above p via m and the second is the ramification index.

`InfinitePlaces(K)`

A sequence containing all the infinite places of the number field K is returned.

`Divisor(pl)`

The divisor $1 * pl$ for a place pl.

`Divisor(I)`

The divisor which is the linear combination of the places corresponding to the factorization of the ideal I and the exponents of that factorization.

`Divisor(x)`

The principal divisor xO where O is the maximal order of the underlying number field of which x is an element. In particular, this computes a finite divisor.

`RealPlaces(K)`

For a number field K a sequence containing all real (infinite) places is computed. For an absolute field this are precisely the embeddings into R coming from the real roots of the defining polynomial.

34.8.4 Arithmetic with Places and Divisors

Divisors and places can be added, negated, subtracted and multiplied and divided by integers.

`d1 + d2` `- d` `d1 - d2` `d * k` `d div k`

34.8.5 Other Functions for Places and Divisors

`Valuation(a, p)`

The valuation of the element a of a number field at the place p.

`Valuation(I, p)`

The valuation of the ideal I at the finite place p.

`Support(D)`

The support of the divisor D as a sequence of places and a sequence of the corresponding exponents.

`Ideal(D)`

The ideal corresponding to the finite part of the divisor D.

`Evaluate(x, p)`

The evaluation of the number field element x in the residue class field of the place p, i.e. for a finite place p this corresponds to the image under the residue class field map for the underlying prime ideal. For infinite places, this returns the corresponding conjugate, ie. a real or complex number.

`RealEmbeddings(a)`

The sequence of real embeddings of the algebraic number a is computed, i.e. a is evaluated at all real places of the number field.

`RealSigns(a)`

A sequence containing ± 1 depending on whether the evaluation of the number field element a at the corresponding real place is positive or negative.

`IsReal(p)`

For an infinite place p, returns `true` if the corresponding embedding is real, i.e. if `Evaluate` at p will give real results.

`IsComplex(p)`

For an infinite place p, return `true` if the corresponding embedding is complex, i.e. if `Evaluate` at p will generally yield complex results.

`IsFinite(p)`

For a place p of a number field, return if the place is finite, i.e. if it corresponds to a prime ideal.

`IsInfinite(p)`

For a place p of a number field return if the place is infinite, ie. if it corresponds to an embedding of the number field into the real or complex numbers. If the place is infinite, the index of the embedding it corresponds to is returned as well.

`Extends(P, p)`

For two places P of K and p of k where K is an extension of k, check whether P extends p. For finite places, this is equivalent to checking if the prime ideal corresponding to P dives, in the maximal order of K the prime ideal of p. For infinite places `true` implies that for elements of k, evaluation at P and p will give identical results.

```
InertiaDegree(P)
Degree(P)
```

For a place P of a number field, return the inertia degree of P. That is for a finite place, return the degree of the residue class field over it's prime field, for infinite places it is always 1.

```
Degree(D)
```

For a divisor D of a number field, the degree is the weighted sum of the degrees of the supporting places, the weights being the multiplicities.

```
NumberField(P)
NumberField(D)
```

For a place P or divisor D of a number field, return the underlying number field.

```
ResidueClassField(P)
```

For a place P of a number field, compute the residue class field of P. For a finite place this will be a finite field, namely the residue class field of the underlying prime ideal. For an infinite place, the residue class field will be the field of real or complex numbers.

```
UniformizingElement(P)
```

For a finite place P of a number field, return an element of valuation 1. This will be the uniformizing element of the underlying prime ideal as well.

```
LocalDegree(P)
```

The degree of the completion at the place P, i.e. the product of the inertia degree times the ramification index.

```
RamificationIndex(P)
```

The ramification index of the place P. For infinite real places this is 1 and 2 for complex places.

```
DecompositionGroup(P)
```

For a place P of a normal number field, return the decomposition group as a subgroup of the (abstract) automorphism group.

34.9 Characters

There is an ability to use characters on number fields, similar to Dirichlet characters over the integers. These are implemented via `DirichletGroup` on number field elements, and `HeckeCharacterGroup` on ideals. The former is the dual of the `RayResidueRing` of an ideal, and the latter is the dual of the `RayClassGroup` of an ideal. Arithmetic on groups can be done multiplicatively, and the characters can be evaluated at suitable field elements and ideals. The `sub` constructor, along with `+` and `meet` for subgroups on the same modulus, should also work.

The associated types are `GrpDrchNF` and `GrpDrchNFElt`, `GrpHecke` and `GrpHeckeElt`. The number field must be an absolute extension of the rationals.

34.9.1 Creation Functions

`DirichletGroup(I)`

`DirichletGroup(I, oo)`

Given an ideal I of the integer ring of the number field K and a set of real places of K, the intrinsic `DirichletGroup` will return the dual group to the `RayResidueRing` of the specified information.

`HeckeCharacterGroup(I)`

`HeckeCharacterGroup(I, oo)`

Given an ideal I of the integer ring of the number field K and a set of real places of K, the intrinsic `HeckeCharacterGroup` will return the dual group to the `RayClassGroup` of the specified information.

`UnitTrivialSubgroup(G)`

Given a group of Dirichlet characters, return the subgroup that is trivial on the image of the field units in the residue ring.

34.9.2 Functions on Groups and Group Elements

`Modulus(G)`

`Modulus(G)`

`Modulus(chi)`

`Modulus(chi)`

Returns the modulus ideal and a (possibly empty) sequence of real places.

`Order(chi)`

`Order(psi)`

Returns the order of a Dirichlet or Hecke character.

`Random(G)`

`Random(G)`

Returns a random element of a Dirichlet or Hecke group.

`Domain(G)`

`Domain(G)`

Returns the number field that is the domain for the Dirichlet character.

`Domain(G)`

`Domain(G)`

Returns the set of ideals that is the domain for the Hecke character.

`Decomposition(chi)`

Returns a list of characters of prime power modulus (and real places) whose product (after extension to the original `DirichletGroup`) is the given Dirichlet character.

`Conductor(chi)`

`Conductor(psi)`

The product of the moduli of the all nontrivial characters in the decomposition of the given Dirichlet character, given as an ideal and a set of real places. Similarly with Hecke characters, where, in fact, one takes the `DirichletRestriction` of the Hecke character, and decomposes this.

`AssociatedPrimitiveCharacter(chi)`

`AssociatedPrimitiveCharacter(psi)`

The primitive Dirichlet character associated to the one that is given, which can be obtained by multiplying all the nontrivial characters in the decomposition. Similarly with Hecke characters, for which this decomposes the `DirichletRestriction` to finds its underlying primitive part, and then takes the `HeckeLift` of this.

`Restrict(chi, D)`

`Restrict(psi, H)`

`Restrict(chi, I)`

`Restrict(psi, I)`

`Restrict(chi, I, oo)`

`Restrict(psi, I, oo)`

`Restrict(G, D)`

`Restrict(G, H)`

`Restrict(G, I)`

`Restrict(G, I)`

`Restrict(G, I, oo)`

`Restrict(G, I, oo)`

Given a Dirichlet character modulo an ideal I and a Dirichlet character group modulo J for which $I \subseteq J$ (including behavior at real places when specified) with the character trivial on $(J/I)^\star$, this returns the restricted character on J. Similarly with Hecke characters, and with an ideal (with possible real places) at the second argument. Also with a group of characters as the first argument.

`TargetRestriction(G, C)`

`TargetRestriction(H, C)`

Given a group of Dirichlet or Hecke characters and a cyclotomic field, return the subgroup of characters whose image is contained in the cyclotomic field.

`SetTargetRing(~chi, e)`

`SetTargetRing(~psi, e)`

Given a Dirichlet or Hecke character and a suitable root of unity, modify the character to take values according to this root unity. The ring element must be a root of unity, and its order must be a multiple of the order of the character. Writing $m = \text{ord}(\chi)$, if the character previously had $\chi(u) = \zeta_m^v$, it will now have $\chi(u) = (e^q)^v$ where q is $\text{ord}(e)/m$.

`Extend(chi, D)`

`Extend(psi, H)`

`Extend(chi, I)`

`Extend(psi, I)`

`Extend(chi, I, oo)`

`Extend(psi, I, oo)`

`Extend(G, D)`

`Extend(G, H)`

`Extend(G, I)`

`Extend(G, I)`

`Extend(G, I, oo)`

`Extend(G, I, oo)`

Given a Dirichlet character modulo I and a Dirichlet character group modulo J for which $J \subseteq I$ (again possibly including the real places), this function returns the induced character on J, that is, the one that is trivial on $(I/J)^\star$. A second return value corresponds to a kernel. Similarly with Hecke characters, and with an ideal (with possible real places) at the second argument. Also with a group of characters as the first argument.

34.9.3 Predicates on Group Elements

`IsTrivial(chi)`

`IsTrivial(psi)`

Returns whether the given character corresponds to the trivial element in the character group.

`IsTrivialOnUnits(chi)`

Returns whether a Dirichlet character is trivial on the units of the number field; this determines whether the character can lift to a Hecke character on the ideals.

`IsOdd(chi)`

Returns whether a Dirichlet character χ has $\chi(-1) = -1$.

`IsEven(chi)`

Returns whether a Dirichlet character χ has $\chi(-1) = +1$.

`IsTotallyEven(chi)`

Returns whether a Dirichlet character χ has $\chi_p(-1) = +1$ for each χ_p in its decomposition.

`IsPrimitive(chi)`

`IsPrimitive(psi)`

Returns whether a Dirichlet character is primitive, that is, whether its conductor and modulus are equal.

34.9.4 Passing between Dirichlet and Hecke Characters

`HeckeLift(chi)`

Given a Dirichlet character that is trivial on the units of the number field, this functions returns a Hecke character that extends its domain to all the ideals of the integer ring. Also returns a kernel, so as to span the set of all possible lifts.

`DirichletRestriction(psi)`

Given a Hecke character on the ideals of the integer ring of a number field, this functions returns the Dirichlet restriction of it on the field elements.

Example H34E16

This example tries to codify the terminology via a standard example with Dirichlet characters over the rationals. We construct various characters modulo 5.

```
> Q := NumberField(Polynomial([-1, 1]) : DoLinearExtension);
> O := IntegerRing(Q);
> I := 5*O;
> DirichletGroup(I);
Abelian Group isomorphic to Z/4
Group of Dirichlet characters of modulus of norm 5 mapping to
 Cyclotomic Field of order 4 and degree 2
```

The above group is the Dirichlet characters modulo 5. However, the odd characters are not characters on ideals, as they are nontrivial on the units. To pass to the Hecke characters, we need to enlarge the modulus to consider embeddings at the real place. Note that this will give four more characters, corresponding to multiplying the above by the character that has χ as $+1$ on positive elements and -1 on negative elements; such characters will not be periodic in the traditional sense of Dirichlet characters, but are still completely multiplicative.

```
> D := DirichletGroup(I, [1]); D; // include first real place
Abelian Group isomorphic to Z/2 + Z/4
Group of Dirichlet characters D of modulus of norm 5 and infinite
places [ 1 ] mapping to Cyclotomic Field of order 4 and degree 2
> [ IsTrivialOnUnits(x) : x in Elements(D) ];
[ true, false, false, true, true, false, false, true ]
>  HeckeLift(D.1); // non-trivial on units
Runtime error in 'HeckeLift': Character is nontrivial on the units
> hl := HeckeLift(D.1 * D.2);
> hl(2);
zeta_4
> hl(2) eq (D.1 * D.2)(2);
true
```

So only half of the 8 completely multiplicative characters on field elements lift to characters on ideals, and these correspond exactly the standard four Dirichlet characters modulo 5, though evinced in a different guise.

`DirichletCharacter(I, B)`

`DirichletCharacter(I, oo, B)`

`DirichletCharacter(G, B)`

`HeckeCharacter(I, B)`

`HeckeCharacter(I, oo, B)`

`HeckeCharacter(G, B)`

`RequireGenerators` BoolElt *Default :* `true`

Given either an ideal (and also possibly a set of real infinite places) or a `DirichletGroup`, and a tuple of 2-tuples each containing a field element and a element of `Integers(m)` for some m, construct a Dirichlet character that sends each field element to the cyclotomic unit corresponding to the residue element. The second member of each 2-tuple can alternatively be an element of some cyclotomic field.

The parameter `RequireGenerators` demands that the given field elements should generate the `RayResidueRing` of the ideal. The second return argument is a subgroup of the ambient `DirichletGroup` by which the returned character can be translated and still retain the same values on the given elements.

Similarly for `HeckeCharacter` – there the first element in each 2-tuple should be an ideal of the field, and `RequireGenerators` demands that these generate the `RayClassGroup` of the ideal.

Example H34E17

We define a character on $5O_K$ that sends $\sqrt{-23}$ to ζ_8^2 (note that $\sqrt{-23}$ has order 8 in the `RayResidueRing` to this modulus).

```
> K := QuadraticField(-23);
> I := 5*IntegerRing(K);
> chi, SG := DirichletCharacter
>           (I, <<K.1, Integers(8)!2>> : RequireGenerators := false);
> chi(K.1);
zeta_4
> (SG.1 * chi)(K.1);
zeta_4
```

And then we define one that sends $\sqrt{-23}$ to ζ_8^6 and $(3 + 2\sqrt{-23})$, an element of order 6 in the `RayResidueRing`, to ζ_{24}^8.

```
> data := <<K.1, Integers(8)!6>, <3+2*K.1,Integers(24)!8>>;
> chi, SG := DirichletCharacter(I, data);
> chi(K.1);
-zeta_4
> chi(3+2*K.1);
zeta_3
> #SG; // this subgroup SG is trivial, as the data determine chi
1
```

Note that we can replace the Integers(8)!6 in the first tuple by ζ_8^6.

```
> C<zeta8> := CyclotomicField(8);
> data2 := <<K.1, zeta8^6>, <3+2*K.1,Integers(24)!8>>;
> chi2 := DirichletCharacter(I, data2);
> chi eq chi2;
```

```
true
```

Now we give a example with Hecke characters over a cubic field. We also note that the evaluation of a character (either Dirichlet or Hecke) can be obtained in "raw" form as an element in a residue ring via the use of the `Raw` vararg.

```
> _<x> := PolynomialRing(Integers());
> K<s> := NumberField(x^3-x^2+7*x-6); // #ClassGroup(K) is 5
> I := Factorization(11*IntegerRing(K))[2][1]; // norm 121
> HG := HeckeCharacterGroup(I,[1]); // has 20 elements
> f3 := Factorization(3*IntegerRing(K))[1][1]; // order 10
> data := < <f3, Integers(10)!7> >;
> psi := HeckeCharacter(HG, data : RequireGenerators := false);
> psi(f3);
-zeta_10^2
> psi(f3) eq CyclotomicField(10).1^7;
true
> '@'(f3,psi : Raw); // get Raw form of evaluation
14
> Parent($1);
Residue class ring of integers modulo 20
> f113 := Factorization(113*IntegerRing(K))[1][1]; // order 4
> data2 := < <f113, Integers(4)!3> >;
> psi := HeckeCharacter(HG, <data[1], data2[1]>);
> psi(f113);
-zeta_4
```

`CentralCharacter(chi)`

`CentralCharacter(psi)`

Given a Dirichlet or Hecke character, compute its central character down to the rationals. This is defined by computing a Dirichlet character that is defined over **Q** and agrees with the given character on a set of generators of the residue ring of the norm of the modulus of the given character. The `AssociatedPrimitiveCharacter` of this is then returned. It should have the same value as the original character on all unramified primes (at least). Note that the central character will always be a Dirichlet character, as the class number of the rationals is 1.

Example H34E18

```
> K := NumberField(Polynomial([4,3,-1,1])); // x^3-x^2+3*x+4
> f7 := Factorization(7*Integers(K))[1][1];
> G:= DirichletGroup(f7^2,[1]);
> chi := G.1*G.2*G.3;
> cc := CentralCharacter(chi); Conductor(cc);
Principal Ideal, Generator: [49] // conductor 49
[ 1 ] // infinite place
```

```
> cc := CentralCharacter(chi^14); Conductor(cc);
Principal Ideal, Generator: [7] // conductor 7
[ ] // no infinite places
> ////////////////
> K := NumberField(Polynomial([-10,-9,-10,1]));
> #ClassGroup(K); // C7 class group
7
> f5 := Factorization(5*Integers(K))[1][1];
> H := HeckeCharacterGroup(f5,[1]);
> cc:=CentralCharacter(H.1); Conductor(cc);
Principal Prime Ideal, Generator: [5] // conductor 5
[ 1 ] // infinite place
> Order(H.1), Order(cc);
28 4
> IsTrivial(CentralCharacter(H.1^4));
true
```

`DirichletCharacterOverNF(chi)`

`DirichletCharacterOverQ(chi)`

These are utility functions to pass between the two types of Dirichlet character over **Q** in Magma. The first takes a Dirichlet character over the `Rationals()` and returns one over the rationals as a number field, and the second reverses this.

Example H34E19

```
> SetSeed(1);
> G := DirichletGroup(16*3^2*5^2*7*11, CyclotomicField(2^6*3*5));
> #G;
57600
> chi := Random(G);
> Order(chi);
30
> psi := DirichletCharacterOverNF(chi);
> Order(psi);
30
> #Parent(psi)`supergroup;
57600
> &and[chi(p) eq psi(p) : p in PrimesUpTo(1000)];
true
> DirichletCharacterOverQ(psi) eq chi;
true
```

34.9.5 L-functions of Hecke Characters

Given a primitive Hecke character ψ, one can define the associated L-function as

$$L(\psi, s) = \prod_{\mathfrak{p}} \bigl(1 - \psi(\mathfrak{p})/\mathrm{N}\mathfrak{p}^s\bigr)^{-1},$$

and this satisfies a functional equation whose conductor is the product of the conductor of ψ and the discriminant of (the integer ring) of the number field for ψ.

Example H34E20

```
> _<x> := PolynomialRing(Integers());
> K<s> := NumberField(x^5 - 2*x^4 + 2*x + 2);
> I2 := Factorization( 2 * IntegerRing(K) ) [1][1]; // ideal above 2
> I11 := Factorization( 11 * IntegerRing(K) ) [1][1]; // above 11
> I := I2*I11; Norm(I);
22
> H := HeckeCharacterGroup(I, [1]);
> #H;
2
> psi := H.1; IsPrimitive(psi);
false
> prim := AssociatedPrimitiveCharacter(psi); Norm(Conductor(prim));
11
> L := LSeries(prim);
> LSetPrecision(L, 10); LCfRequired(L); // number of terms needed
4042
> CheckFunctionalEquation(L); // 2 or 3 seconds
-3.492459655E-10
```

34.9.6 Hecke Grössencharacters and their L-functions

A limited ability to compute Grössencharacters and their L-functions exists, but it has only really been tested for complex quadratic fields and a bit for $\mathbf{Q}(\zeta_5)$. These are "quasi-characters" in that their image is not restricted to the unit circle (and 0). The implementation in Magma handles the "algebraic" characters of this sort, or those of type A_0; it also requires that the field of definition be a CM-field (an imaginary quadratic extension of a totally real field).

The natural definition of Grössencharacters would be on a coset of the dual group of the `RayResidueRing` extended by the `ClassGroup`, without any modding out by units (which gives the `RayClassGroup`). However, the Magma implementation uses a Hecke character combined with an auxiliary Dirichlet character to simulate this. Arithmetic with Grössencharacters in also possible, even though there is no underlying group structure. However, equality with Grössencharacters is not implemented (one needs to check that various class group representatives are compatible, etc.).

```
Grossencharacter(psi, chi, T)
```

```
RawEval(I, GR)
```

Given a Hecke character ψ and a Dirichlet character χ (of the same modulus) and a compatible ∞-type T return the associated Grössencharacter.

The ∞-type is a sequence of pairs of integers which correspond to embeddings, such that

$$\psi\big((\alpha)\big) = \prod_{i=1}^{\#T} (\alpha^{\sigma_i})^{T[i][1]} (\bar{\alpha}^{\sigma_i})^{T[i][2]}$$

for all α that are congruent to 1 modulo the modulus of ψ. For a Grössencharacter to exist it follows that the ∞-type must trivialise all (totally positive) units that are congruent to 1 modulo the modulus of ψ. Each pair in T must have the same sum.

The Dirichlet character χ must correspond to the action on the image of the `UnitGroup` in the `RayResidueRing` of the modulus. In particular, for every unit u we must have that

$$\chi(u) = \prod_{i=1}^{\#T} (u^{\sigma_i})^{T[i][1]} (\bar{u}^{\sigma_i})^{T[i][2]},$$

and since the ∞-type is multiplicative, we need only check this on generators of the units.

Evaluating a Grössencharacter returns a complex number, corresponding to some choice of internal embeddings. The use of `RawEval` on an ideal will return an element in an extension of the field K (to which the ∞-type is then applied) and two elements in cyclotomic fields, corresponding to evaluations for χ and ψ respectively.

```
Grossencharacter(psi, T)
```

Same as above, but Magma will try to compute a compatible Dirichlet character χ for the given data. If there is more than one possibility, an arbitrary choice could be made.

```
Conductor(psi)
```

```
Modulus(psi)
```

```
IsPrimitive(psi)
```

```
AssociatedPrimitiveGrossencharacter(psi)
```

The conductor of the Grössencharacter is the conductor of the product of the Dirichlet part with the `DirichletRestriction` of the Hecke part. A Grössencharacter is primitive if its modulus is the same as the conductor. When taking L-functions, as before the conductor is multiplied by the discriminant of the integer ring of the field.

```
Extend(psi, I)
```

```
Restrict(psi, I)
```

Extension and restriction of a Grössencharacter. Note that the second argument is an ideal, unlike the Dirichlet/Hecke cases, where it is a group of characters.

```
CentralCharacter(psi)
```

Compute the central character (down to $\mathbf{Q}$) of a Grössencharacter, normalizing the result to be weight 0, and returning it as a Dirichlet character (over $\mathbf{Q}$ as a number field).

Example H34E21

First, an example of $[1, 0]$-type Grössencharacters on the Gaussian field, with modulus $\mathfrak{p}_2^3$ where $\mathfrak{p}_2$ is the (ramified) prime above 2. This induces the L-function for the congruent number curve.

```
> K<i> := QuadraticField(-1);
> I := (1+i)^3*IntegerRing(K);
> HG := HeckeCharacterGroup(I, []);
> DG := DirichletGroup(I, []); #DG;
4
> GR := Grossencharacter(HG.0, DG.1^3, [[1,0]]);
> L := LSeries(GR); CheckFunctionalEquation(L);
1.57772181044202361082345713057E-30
> CentralValue(L);
0.655514388573029952616209897475
> CentralValue(LSeries(EllipticCurve("32a")));
0.655514388573029952616209897473
```

Example H34E22

An example with the canonical Grössencharacter for $K = \mathbf{Q}(\sqrt{-23})$. The ramification here is only at the prime above 23.

```
> K<s> := QuadraticField(-23);
> I := Factorization(23*IntegerRing(K))[1][1]; // ramified place
> HG := HeckeCharacterGroup(I, []);
> DG := DirichletGroup(I, []); #DG;
22
> GR := Grossencharacter(HG.0, DG.1^11, [[1,0]]); // canonical character
> CheckFunctionalEquation(LSeries(GR));
-5.17492753824983744350093938826E-28
> H := K`extension_field; H; // defined by internal code
Number Field with defining polynomial y^3 + 1/2*(s + 3) over K
```

The values of the Grössencharacter are in the given field extension of K. We can also twist the Grössencharacter by a Hecke character on I, either via the `Grossencharacter` intrinsic, or by direct multiplication.

```
> i2 :=  Factorization(2*IntegerRing(K))[1][1]; // ideal of norm 2
> (GR*HG.1)(i2); // evaluation at i2
-0.140157638956246665944180880120 - 1.40725116316960648195556086783*i
> GR2 := Grossencharacter(HG.1, DG.1^11, [[1,0]]); // psi over zeta_11
> GR2(i2);
-0.140157638956246665944180880120 - 1.40725116316960648195556086783*i
> RawEval(i2,GR2);  // first value is in the cubic extension of K
H.1
1
zeta_33^4
> CheckFunctionalEquation(LSeries(GR2));
-7.09974814698910624870555708755E-30
```

Example H34E23

An example from Fernando Rodriguez Villegas where the Grössencharacter yields an L-function with even functional equation, but vanishing central value. This is the cube of the canonical character on $\mathbf{Q}(\sqrt{-59})$, which has class number 3. The L-function can be alternatively realised from a weight 4 modular form of level 59^2.

```
> K := QuadraticField(-59);
> I := Factorization(59*IntegerRing(K))[1][1];
> H := HeckeCharacterGroup(I);
> DG := DirichletGroup(I);
> GR := Grossencharacter(H.0, DG.1^29, [[3,0]]); // cube of canonical char
> L := LSeries(GR);
> CheckFunctionalEquation(L);
0.000000000000000000000000000000
> Sign(L);
1.00000000000000000000000000000
> CentralValue(L);
3.51858026759864075475017925650E-30
> LSetPrecision(L, 9);
> LTaylor(L, 2, 3); // first 3 terms of Taylor series about s=2
-1.09144041E-12 + 9.82515510E-11*z + 2.87637101*z^2 - 7.65817878*z^3 + ...
```

The same Grössencharacter can be obtained from cubing the canonical character (of type $[1,0]$).

```
> GR3 := Grossencharacter(H.0, DG.1^29, [[1,0]])^3;
> CentralValue(LSeries(GR3));
3.51858026759864075475017925650E-30
```

Example H34E24

An example with $\mathbf{Q}(\zeta_5)$, comparing the central value to the periods derived from Γ-values.

```
> _<x> := PolynomialRing(Rationals());
> K<z5> := NumberField(x^4+x^3+x^2+x+1);
> p5 := Factorization(5*IntegerRing(K))[1][1]; // ramified prime above 5
> H := HeckeCharacterGroup(p5^2);
> DG := DirichletGroup(p5^2);  // need p5^2 to get chi with this oo-type
> chi := DG.1^2*DG.2; // could alternatively have Magma compute this
> GR := Grossencharacter(H.0, chi, [[3,0],[1,2]]);
```

We can compute that this ∞-type sends ζ_5 to $(\zeta_5^1)^3 \cdot (\zeta_5^4)^0 \cdot (\zeta_5^2)^1 \cdot (\zeta_5^3)^2 = \zeta_5^{11}$ under the default embedding, and thus the ideal needs to afford a character of order 5 for a Grössencharacter to exist.

```
> L := LSeries(GR);
> LSeriesData(L); // Conductor is Norm(p5^2) * disc(K) = 5^2 * 5^3
<4, [ 0, -1, 1, 0 ], 3125, ... >;
> CheckFunctionalEquation(L); // functional equation works
0.000000000000000000000000000000
> CentralValue(L); // same as Evaluate(L,2)
1.25684568045898366613593980559
> Gamma(1/5)^3 * Gamma(2/5)^3 / Gamma(3/5)^2 / Gamma(4/5)^2 / 5^(7/2);
1.25684568045898366613593980558
```

The $[[3,0],[2,1]]$ ∞-type sends ζ_5 to $\zeta_5^{3+0+2\cdot 2+3} = 1$, but we still need $\mathfrak{p}_5$ in the modulus to trivialise the units of infinite order.

```
> H := HeckeCharacterGroup( 1 * IntegerRing(K)); // try conductor 1
>  GR := Grossencharacter(H.0, [[3,0],[2,1]]);
Runtime error in 'Grossencharacter':
oo-type should be trivial on all totally positive units that are 1 mod I
Fails for -zeta_5^2 - 1 which gives -1.000000000 - 3.293785801E-101*$.1
> H := HeckeCharacterGroup(p5); // conductor of norm 5
> GR := Grossencharacter(H.0, [[3,0],[2,1]]); // finds a character
> L := LSeries(GR); // CheckFunctionalEquation(L);
> PI := Pi(RealField());
> CentralValue(L); // now recognise as a product via logs and LLL
0.749859246433372123005585683300
> A := [ Gamma(1/5), Gamma(2/5), Gamma(3/5), Gamma(4/5), 5, PI, $1 ];
> LOGS := [ ComplexField() ! Log(x) : x in A ];
> LinearRelation(LOGS : Al := "LLL");
[ -14, 2, -2, 14, 15, -4, 4 ]
```

Example H34E25

Twisting a Grössencharacter by a Hilbert character is equivalent to changing the embedding.

```
> K := QuadraticField(-39);
> I := 39*IntegerRing(K);
> F := &*[f[1] : f in Factorization(I)]; // ideal of norm 39
> H := HeckeCharacterGroup(F); H;
Abelian Group isomorphic to Z/4 + Z/12 given as Z/4 + Z/12
> Norm(Conductor(H.1)); // H.1 is a Hilbert character of norm 1
1
```

```
> GR := Grossencharacter(H.0, [[3,0]]); // third power
```

There are four Hilbert characters here (from the class group of K), and we twist the Grössencharacter by each.

```
> L0 := LSeries(AssociatedPrimitiveGrossencharacter(GR));
> L1 := LSeries(AssociatedPrimitiveGrossencharacter(GR*H.1));
> L2 := LSeries(AssociatedPrimitiveGrossencharacter(GR*H.1^2));
> L3 := LSeries(AssociatedPrimitiveGrossencharacter(GR*H.1^3));
> Ls := [ L0, L1, L2, L3 ]; for L in Ls do LSetPrecision(L, 10); end for;
> for L in Ls do [CentralValue(L), Sign(L)]; end for;
[ 1.335826177, 1.000000000 + 2.706585223E-10*i ]
[ -1.373433032*i, -0.9999999999 - 6.351223882E-11*i ]
[ 1.335826177, 1.000000000 - 2.706585223E-10*i ]
[ 1.373433032*i, -0.9999999999 + 6.351223882E-11*i ]
```

The embedding information is stored internally in `K'Hip`, and we modify this directly to get the same L-values via a different method.

```
> K`Hip; // extension of infinite place of K
[ [ 1, 1 ] place at infinity ]
> IP := InfinitePlaces(K`extension_field); IP;
[ [ 1, 1 ] place at infinity, [ 1, 2 ] place at infinity,
  [ 1, 3 ] place at infinity, [ 1, 4 ] place at infinity ]
> for ip in IP do K`Hip := [ ip ]; // change ip, but use same GR
>       L := LSeries(AssociatedPrimitiveGrossencharacter(GR));
>       LSetPrecision(L, 10); [CentralValue(L), Sign(L)]; end for;
[ 1.335826177, 1.000000000 - 2.706585223E-10*i ]
[ 1.373433032*i, -0.9999999999 + 6.351223882E-11*i ]
[ 1.335826177, 1.000000000 - 2.706585223E-10*i ]
[ -1.373433032*i, -0.9999999999 - 6.351223882E-11*i ]
```

Finally, we can note that all the Hilbert characters have sign $+1$ in their functional equations, though two of the twists of the Grössencharacter have sign -1.

```
> Ls := [ LSeries(AssociatedPrimitiveCharacter(H.1^k)) : k in [1..4] ];
> [ Sign(L) where _:=CheckFunctionalEquation(L) : L in Ls ];
[ 0.999999999999999999999999999997, 1.00000000000000000000000000000,
  0.999999999999999999999999999997, 1.00000000000000000000000000000 ]
```

Example H34E26

A final example with characters of trivial conductor, here of type $[2,0]$ in $\mathbf{Q}(\sqrt{-23})$.

```
> K<s> := QuadraticField(-23); // class number 3
> I := 1*IntegerRing(K);
> HG := HeckeCharacterGroup(I, []);
> GR := Grossencharacter(HG.0, [[2,0]]); // of oo-type (2,0)
> Evaluate(LSeries(GR), 2); // value at edge of critical strip
1.23819100212426040400794384795
> Evaluate(LSeries(GR*HG.1), 2); // twist by nontrivial Hecke char
```

```
0.670337208665839403747922477469
> Evaluate(LSeries(GR*HG.1^2), 2);
1.061105832664497283099074059600
```

The product of these three L-values should be related to values of the Γ-function at $k/23$ for integral k. (One could alternatively relate these L-values to periods of an elliptic curve over the Hilbert class field of K having ramification only above 23.)

```
> SetDefaultRealFieldPrecision(100);
> e1 := Evaluate(LSeries(GR : Precision:=100), 2);
> e2 := Evaluate(LSeries(GR*HG.1 : Precision:=100), 2);
> e3 := Evaluate(LSeries(GR*HG.1^2 : Precision:=100), 2);
> GAMMA := [Gamma(i/23) : i in [1..22]];
> A := GAMMA cat [3,23,Pi(RealField())] cat [e1,e2,e3];
> LOGS := [ComplexField()!Log(x) : x in A];
> LinearRelation(LOGS : Al:="LLL");
[ 2,  2,  2,  2, -2,  2, -2,  2,  2, -2, -2,
  2,  2, -2, -2,  2, -2,  2, -2, -2, -2, -2,
 -2, -7,  6, -2, -2, -2 ]
> &*[ GAMMA[i]^(2*(DirichletGroup(23).1)(i)) : i in [1..22] ];
24723927.96264290790447830542942451626433185347196157309315591128
> 3^2 * 23^7 / Pi(RealField())^6 * (e1*e2*e3)^2;
24723927.96264290790447830542942451626433185347196157309315591128
```

Example H34E27

An example from [vGvS93, §8.8]. Here the Grössencharacter is on an ideal of norm 2^4 in the cyclotomic field $\mathbf{Q}(\zeta_8)$. The Euler factors will factor in various fields. In Table 7.6 of the cited paper, one notes that the coefficients satisfy $a_{17} = -180$ and $a_{17^2} = 15878$.

```
> Q<z8> := CyclotomicField(8);
> p2 := Factorization(2*Integers(Q))[1][1];
> G := HeckeCharacterGroup(p2^4);
> psi := G.0; // trivial
> GR := Grossencharacter(psi, [[3,0],[1,2]]);
> L:=LSeries(GR);
> CheckFunctionalEquation(L);
6.31088724176809444329382852226E-30
> Factorization(EulerFactor(L,7 : Integral)); // p is 7 mod 8
[ <343*x^2 + 1, 2> ]
> K<s2> := QuadraticField(-2);
> _<t> := PolynomialRing(K);
> Factorization(EulerFactor(L,3 : Integral),K); // 3 mod 8
[ <t^2 + 1/81*(-2*s2 - 1), 1>, <t^2 + 1/81*(2*s2 - 1), 1> ]
> K<i> := QuadraticField(-1);
> _<t> := PolynomialRing(K);
> Factorization(EulerFactor(L,5 : Integral),K); // 5 mod 8
[ <t^2 + 1/3125*(-24*i + 7), 1>, <t^2 + 1/3125*(24*i + 7), 1> ]
> EulerFactor(L,17 : Integral); // -180 and 15878 as desired
```

```
24137569*x^4 - 884340*x^3 + 15878*x^2 - 180*x + 1
```

34.10 Number Field Database

An optional database of number fields may be downloaded from the Magma website. This section defines the interface to that database.

There are databases for number fields of degrees 2 through 9. In the case of degree 2 the enumeration is complete in the discriminant range (discriminants of absolute value less than a million); the other databases include fields with small (absolute value of) discriminant, as well as various other fields that may be of interest. The selection of fields is eclectic, and it may well be that certain "obvious" ones are missing.

For each number field in the database, the following information is stored and may be used to limit the number fields of interest via the `sub` constructor: The discriminant; the signature; the Galois group; the class number; and the class group.

34.10.1 Creation

`NumberFieldDatabase(d)`

Returns a database object for the number fields of degree d.

`sub< D | dmin, dmax : parameters >`

`sub< D | dabs : parameters >`

`sub< D | : parameters >`

`Signature`	[RngIntElt]	*Default :*
`Signatures`	[[RngIntElt]]	*Default :*
`GaloisGroup`	RngIntElt *or* GrpPerm	*Default :*
`ClassNumber`	RngIntElt	*Default :*
`ClassNumberLowerBound`	RngIntElt	*Default :* 1
`ClassNumberUpperBound`	RngIntElt	*Default :* ∞
`ClassNumberBounds`	[RngIntElt]	*Default :* $[1, \infty]$
`ClassGroup`	[RngIntElt]	*Default :*
`ClassSubGroup`	[RngIntElt]	*Default :* [ ]
`ClassSuperGroup`	[RngIntElt]	*Default :*
`SearchByValue`	BoolElt	*Default :* `false`

Returns a sub-database of D, restricting (or further restricting, if D is already a sub-database of the full database for that degree) the contents to those number fields satisfying the specified conditions. Note that (with the exception of `SearchByValue`, which does not actually limit the fields in the database) it is not possible to "undo" restrictions with this constructor — the results are always at least as limited as D is.

The valid non-parameter arguments are up to two integers specifying the desired range of discriminants in the result. If two integers are provided these are taken as the lower (`dmin`) and upper (`dmax`) bounds on the discriminant. If one integer is provided it is taken as a bound (`dabs`) on the absolute value of the discriminant. If no integers are provided then the discriminant range is the same as for D.

The parameters `Signature` or `Signatures` may be used to specify the desired signature or signatures to match. A signature is specified as a sequence of two integers $[s_1, s_2]$ where $s_1 + 2s_2 = d$.

The parameter `GaloisGroup` may be used to specify the desired Galois group of the number field. It may be given as either a permutation group, or as the explicit index of this Galois group in the transitive groups database. (i.e., the first return value of `TransitiveGroupIdentification`.)

It is possible to require certain divisibility conditions on the class number. Internally, there are lower and upper bounds on this value, such that a number field will only match if its class number is divisible by the lower bound and divides the upper bound. These bounds may be set individually using `ClassNumberLowerBound` or `ClassNumberUpperBound`; setting `ClassNumber` is equivalent to setting both bounds to the same value. Both values may be specified at once by setting `ClassNumberBounds` to the sequence of the lower and upper bounds.

When finer control is desired, it is possible to specify desired sub- and supergroups of the class group. Each group is specified by the sequence of its Abelian invariants; the desired subgroup is set using `ClassSubGroup`, and the desired supergroup is set using `ClassSuperGroup`. Both may be set at once (thus requiring the group to match exactly) using `ClassGroup`.

When iterating through the database, the default is to iterate in order of the absolute value of the discriminant. Sometimes it is desirable to iterate in order of the actual value of the discriminant; this can be accomplished by setting the parameter `SearchByValue` to true.

34.10.2 Access

```
Degree(D)
```

Returns the degree of the number fields stored in the database.

```
DiscriminantRange(D)
```

Returns the smallest and largest discriminants of the number fields stored in the database.

```
#D
NumberOfFields(D)
```

Returns how many number fields are stored in the database.

`NumberOfFields(D, d)`

Returns how many number fields of discriminant d are stored in the database.

`NumberFields(D)`

Returns the sequence of number fields stored in the database.

`NumberFields(D, d)`

Returns the sequence of number fields of discriminant d stored in the database.

Example H34E28

We illustrate with some basic examples. We start with the degree 2 number fields and get some basic information about the database.

```
> D := NumberFieldDatabase(2);
> DiscriminantRange(D);
-999995 999997
> #D;
607925
```

There are 13 fields with discriminant of absolute value less than 20:

```
> D20 := sub<D | 20>;
> #D20;
13
> [ Discriminant(F) : F in D20 ];
[ -3, -4, 5, -7, -8, 8, -11, 12, 13, -15, 17, -19, -20 ]
```

Note that these were listed in order of absolute value of the discriminant; we can also list them in value order:

```
> [ Discriminant(F) : F in sub<D20 | : SearchByValue> ];
[ -20, -19, -15, -11, -8, -7, -4, -3, 5, 8, 12, 13, 17 ]
```

Two of these number fields have class number 2:

```
> NumberFields(sub<D20 | : ClassNumber := 2>);
[
    Number Field with defining polynomial x^2 + x + 4 over the Rational Field,
    Number Field with defining polynomial x^2 + 5 over the Rational Field
]
```

Example H34E29

Now we move onto the fields of degree five, and in particular those with signature [1,2].

```
> D := NumberFieldDatabase(5);
> #D;
289040
> D12 := sub<D |: Signature := [1,2]>;
> #D12;
```

```
186906
```

We consider how many of these have class number equal to four. Note the cumulative nature of the restrictions has come into play.

```
> #sub<D12 |: ClassNumber := 4>;
1222
> #sub<D |: ClassNumber := 4>;
1255
```

The number with class group specifically isomorphic to $C_2 \times C_2$ is much less:

```
> #sub<D12 |: ClassGroup := [2,2]>;
99
```

34.11 Bibliography

[**Bai96**] Georg Baier. Zum Round 4 Algorithmus. Diplomarbeit, Technische Universität Berlin, 1996. URL:http://www.math.tu-berlin.de/~kant/publications/diplom/baier.ps.gz.

[**Bia**] J.-F. Biasse. Number field sieve to compute Class groups.

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics.* Springer, Berlin–Heidelberg–New York, 1993.

[**Coh00**] Henri Cohen. *Advanced Topics in Computational Number Theory.* Springer, Berlin–Heidelberg–New York, 2000.

[**Fie97**] Claus Fieker. *Über relative Normgleichungen in algebraischen Zahlkörpern.* Dissertation, Technische Universität Berlin, 1997. URL:http://www.math.tu-berlin.de/~kant/publications/diss/diss_CF.ps.gz.

[**Fri97**] Carsten Friedrichs. Berechnung relativer Ganzheitsbasen mit dem Round-2-Algorithmus. Diplomarbeit, Technische Universität Berlin, 1997. URL:http://www.math.tu-berlin.de/~kant/publications/diplom/friedrichs.ps.gz.

[**Gar80**] Dennis A. Garbanati. An Algorithm for finding an algebraic number whose norm is a given rational number. *J. reine angew. Math.*, 316:1–13, 1980.

[**Jac99**] M. J. Jacobson, Jr. Applying sieving to the computation of quadratic class groups. *Math. Comp.*, 68(226):859–867, 1999.

[**KAN97**] KANT Group. KANT V4. *J. Symbolic Comp.*, 24(3–4):267–383, 1997.

[**KAN00**] KANT Group. The Number Theory Package KANT/KASH. URL:http://www.math.tu-berlin.de/~kant, 2000.

[**Pau01**] Sebastian Pauli. Factoring polynomials over local fields. *Journal of Symbolic Computation*, 32(5):533–547, 2001.

[**Poh93**] M. Pohst. *Computational Algebraic Number Theory.* DMV Seminar Band 21. Birkhäuser Verlag, Basel - Boston - Berlin, 1993.

[**PZ89**] Michael E. Pohst and Hans Zassenhaus. *Algorithmic Algebraic Number Theory*. Encyclopaedia of mathematics and its applications. Cambridge University Press, Cambridge, 1989.

[**Sim02**] Denis Simon. Solving norm equations in relative number fields using S-units. *Math. Comput.*, 71(239):1287–1305, 2002.

[**Tra76**] Barry M. Trager. Algebraic factoring and rational function integration. In R.D. Jenks, editor, *Proc. SYMSAC '76*, pages 196–208. ACM press, 1976.

[**vGvS93**] B. van Geemen and D. van Straten. The cusp forms of weight 3 on $\Gamma_2(2,4,8)$. *Math. Comp.*, 61(204):849–872, 1993.

35 QUADRATIC FIELDS

Chapter 35
QUADRATIC FIELDS

35.1 Introduction

Quadratic fields in MAGMA can be created as a subtype of the number fields `FldNum`. The advantage of the special quadratic fields is that some special (faster) algorithms have been or will be implemented to deal with them; the functions for the special quadratic fields (created with the `QuadraticField` function) are described here. Functions which work generally for number fields and their orders are described in Chapter 34.

The categories involved are `FldQuad` for fields, `RngQuad` for their orders and `FldQuadElt` and `RngQuadElt` for their elements.

35.1.1 Representation

For every squarefree integer d (not 0 or 1) there is a unique quadratic field $\mathbf{Q}(\sqrt{d})$; for any integer k we have the field $\mathbf{Q}(\sqrt{k^2 d}) \cong \mathbf{Q}(\sqrt{d})$. Given any integer m, the function `QuadraticField` will create a structure corresponding to the quadratic field $\mathbf{Q}(\sqrt{d})$, where d is the squarefree kernel of m (d will have the same sign as m and its absolute value is the largest squarefree divisor of m). In MAGMA a list of quadratic fields currently present is maintained, and if $\mathbf{Q}(\sqrt{d})$ has been created before a reference on it will be returned: two fields with the same d are the same. The discriminant D of $\mathbf{Q}(\sqrt{d})$ will be $D = d$ if $d \equiv 1 \bmod 4$ and $D = 4d$ if $d \equiv 2, 3 \bmod 4$.

Elements of $\mathbf{Q}(\sqrt{d})$ are represented by a common positive denominator b and two integer coefficients: $\alpha = \frac{1}{b}(x + y\sqrt{d})$.

The ring of integers of $F = \mathbf{Q}(\sqrt{d})$ will be $O_F = \mathbf{Z} + \epsilon_d \mathbf{Z}$, where

$$\epsilon_d = \begin{cases} \sqrt{d} & \text{if } d \equiv 2, 3 \bmod 4, \\ \frac{1+\sqrt{d}}{2} & \text{if } d \equiv 1 \bmod 4. \end{cases}$$

Elements of O_F are represented by two integer coefficients $\alpha = x + y\epsilon_d$. The pair $1, \epsilon_d$ forms an integral basis for $F = \mathbf{Q}(\sqrt{d})$, but note that elements of F are represented using the basis of the equation order $(1, \sqrt{d})$ instead.

For any positive integer f there is a suborder of conductor f in O_F, whose elements are of the form $x + yf\epsilon_d$, for any integers x, y. The discriminant of the order of conductor f is $f^2 D$, where D is the field discriminant.

The equation order of F is $E_F = \mathbf{Z} + \sqrt{d}\mathbf{Z}$. Suborders of conductor f can be formed which will contain elements of the form $x + yf\sqrt{d}$ for any integers x and y.

35.2 Creation of Structures

Squarefree integers determine quadratic fields. Associated with any quadratic field is its ring of integers (maximal order) and an equation order, and for every positive integer f there exists an order of conductor f inside the maximal order.

For information on creating elements see Section 34.2.3.

```
QuadraticField(m)
```

Given an integer m that is not a square, create the field $\mathbf{Q}(\sqrt{d})$, where d is the squarefree part of m. It is possible to assign a name to $\sqrt{d}$ using angle brackets: `R<s> := QuadraticField(m)`.

```
EquationOrder(F)
```

Creation of the order $\mathbf{Z}[\sqrt{d}]$ in the quadratic field $F = \mathbf{Q}(\sqrt{d})$, with d squarefree.

```
MaximalOrder(F)
IntegerRing(F)
RingOfIntegers(F)
```

Given a quadratic field $F = \mathbf{Q}(\sqrt{d})$, with d squarefree, create its maximal order. This order is $\mathbf{Z}[\sqrt{d}]$ if $d \equiv 2, 3 \bmod 4$ and $\mathbf{Z}[\frac{1+\sqrt{d}}{2}]$ if $d \equiv 1 \bmod 4$.

```
NumberField(O)
```

Given a quadratic order, this returns the quadratic field of which it is an order.

```
sub<  O | f  >
```

Create the sub-order of index f in the order O of a quadratic field. If O is maximal, this will be the unique order of conductor f.

```
IsQuadratic(K)
IsQuadratic(O)
```

Return `true` if the field K or order O can be created as a quadratic field or order and the quadratic field or order if so.

Example H35E1

We create the quadratic field $\mathbf{Q}(\sqrt{5})$ and an order in it, and display some elements of the order in their representation as order element and as field element.

```
> Q<z> := QuadraticField(5);
> Q eq QuadraticField(45);
true
> O<w> := sub< MaximalOrder(Q) | 7 >;
> O;
Order of conductor 7 in Q
> w;
```

```
w
> Q ! w;
1/2*(7*z + 7)
> Eltseq(w), Eltseq(Q ! w);
[ 0, 1 ]
[ 7/2, 7/2 ]
> ( (7/2)+(7/2)*z )^2;
1/2*(49*z + 147)
>  Q ! w^2;
1/2*(49*z + 147)
> w^2;
7*w + 49
```

Example H35E2

We define an injection $\phi : \mathbf{Q}(\sqrt{5}) \to \mathbf{Q}(\zeta_5)$. First a square root of 5 is identified in $\mathbf{Q}(\zeta_5)$.

```
> Q<w> := QuadraticField(5);
> F<z> := CyclotomicField(5);
> C<c> := PolynomialRing(F);
> Factorization(c^2-5);
[
    <c - 2*z^3 - 2*z^2 - 1, 1>,
    <c + 2*z^3 + 2*z^2 + 1, 1>
]
> h := hom< Q -> F | -2*z^3 - 2*z^2 - 1 >;
> h(w)^2;
5
```

35.3 Operations on Structures

The majority of functions for quadratic fields and orders apply identically to number fields and orders in general. The functions which exist only for quadratic fields and orders are listed here along with those which deserve a special mention.

`AssignNames(~F, [s])`

`AssignNames(~O, [s])`

Procedure to change the name of the generator of a quadratic field F or an order O in a quadratic field to the string s. Elements of the quadratic field $\mathbf{Q}(\sqrt{d})$ with m squarefree will be printed in the form `1/`b`*(`x` + `y`*s)`, where b, x, y are integers. Similarly, for an order O of conductor f in a quadratic field elements will be printed in the format x `+` y`*s`.

This procedure only changes the name used in printing the elements of F or O, it does *not* make an assignment to an identifier s. To do this, use an assignment statement, or angle brackets when creating the field or order: `F<s> := QuadraticField(-3);`.

Note that since this is a procedure that modifies F or O, it is necessary to have a reference $\sim$ in the call to this function.

`Name(F, 1)`

`Name(O, 1)`

Given a quadratic field F or one of its orders O, return the element which has the name attached to it, that is, return $\sqrt{d}$ in the field, or $f\epsilon_d$ in a suborder of the maximal order or $f\sqrt{d}$ in a suborder of the equation order.

`FundamentalUnit(K)`

`FundamentalUnit(O)`

A generator for the unit group of the order O or the maximal order of the quadratic field K.

`Discriminant(K)`

The discriminant of the field K which is only defined up to squares. The discriminant will be the discriminant of the polynomial or better.

`Conductor(K)`

The conductor of the field K which is the order of the smallest cyclotomic field containing K and a sequence containing the ramified real places of K.

`Conductor(O)`

The conductor of the order O, which equals the index of O in the maximal order.

35.3.1 Ideal Class Group

The function `ClassGroup` is available for number fields and orders in general but a different and faster algorithm is used by default for the quadratics. All of the algorithms, except for the sieving method described in [Jac99] which uses the multiple polynomial quadratic sieve (MPQS), are based on binary quadratic forms, see `ClassGroup` on page 758 in Chapter 33 for details.

`ClassGroup(K)`

`ClassGroup(O)`

`FactorBasisBound`	FldReElt	*Default :* 0.1
`ProofBound`	FldReElt	*Default :* 6
`ExtraRelations`	RngIntElt	*Default :* 1
`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`ClassGroupSieve`	*Maximum :* 5

The class group of a maximal order O or the maximal order of the quadratic field K, as an abelian group. The function also returns a map between the group and the power structure of ideals of O or the maximal order of K. The parameter `Al` can be set to `"Sieve"` or `"NoSieve"` to control whether the sieving algorithm is used or not; by default it is used when the discriminant is greater than 10^{20}. For more details on the parameters see `ClassGroup` on page 758 in Chapter 33.

`ClassNumber(K)`

`ClassNumber(O)`

`FactorBasisBound`	FldReElt	*Default :* 0.1
`ProofBound`	FldReElt	*Default :* 6
`ExtraRelations`	RngIntElt	*Default :* 1
`Al`	MonStgElt	*Default :* "*Automatic*"

The class number of the maximal order O or the maximal order of the quadratic field K.

`PicardGroup(O)`

`PicardNumber(O)`

`FactorBasisBound`	FldReElt	*Default :* 0.1
`ProofBound`	FldReElt	*Default :* 6
`ExtraRelations`	RngIntElt	*Default :* 1
`Al`	MonStgElt	*Default :* "*Automatic*"

The picard group (the group of the invertible ideals of O modulo the principal ones) of the order O or the size of this group. `PicardGroup` also returns a map from the group to the ideals of O.

Example H35E3

We give examples of class group calculations using sieving. We also show the use of the mapping between the group and the set of ideals. First a field with negative discriminant.

```
> D:=-(10^(30) + 3 );
> K := QuadraticField(D);
> time G, m := ClassGroup(K : Al := "Sieve");
Time: 10.750
> G, m;
Abelian Group isomorphic to Z/125355959329602
Defined on 1 generator
Relations:
    125355959329602*G.1 = 0
Mapping from: GrpAb: G to Set of ideals of Maximal Order of K
> m(G.1);
Ideal
Two element generators:
    385706622580333
    1487695987023274464670383908888*$.2 + 307255216496036
> $1 @@ m;
G.1
```

And now a field with positive discriminant.

```
> K := QuadraticField(NextPrime(10^24));
> time G, m := ClassGroup(K : Al := "Sieve");
Time: 3.330
> G, m;
Abelian Group isomorphic to Z/3
Defined on 1 generator
Relations:
    3*$.1 = 0
Mapping from: GrpAb: G to Set of ideals of Maximal Equation Order of K
> m(G.1);
Ideal
Two element generators:
    3847
    $.2 + 616
> $1 @@ m;
G.1
> IsPrincipal($2^3);
true
```

`QuadraticClassGroupTwoPart(K)`

`QuadraticClassGroupTwoPart(O)`

`QuadraticClassGroupTwoPart(d)`

`Factorization` RngIntEltFact *Default :* []

Use the Bosma-Stevenhagen algorithm to compute the 2-part of the class group of a quadratic order. Returned are: an array of forms that generates the 2-part and an array that gives the orders of the respective elements. The Factorization of the given discriminant can be given as additional information.

Example H35E4

```
> G, f := QuadraticClassGroupTwoPart(33923894057872); G;
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4 + Z/16 + Z/16
> Random(G);
11*G.1 + 2*G.2 + G.3 + G.4 + G.6
> f($1);
<-1212992,3947508,3780131>
> G, f := QuadraticClassGroupTwoPart(QuadraticField(33923894057872)); G;
Abelian Group isomorphic to Z/2 + Z/8 + Z/16
```

35.3.2 Norm Equations

For imaginary quadratic fields, (that is, for quadratic fields $\mathbf{Q}(\sqrt{m})$ with $m < 0$), the function `NormEquation` is provided specially for quadratics to find integral elements of a given norm. For real quadratic fields conics are used, see Section 119.5.1 for details.

`NormEquation(F, m)`

`NormEquation(F, m: parameters)`

`NormEquation(O, m)`

`NormEquation(O, m: parameters)`

`Factorization`	[<RngIntElt, RngIntElt>]	
`All`	BoolElt	*Default :* `true`
`Solutions`	RngIntElt	*Default :* All
`Exact`	BoolElt	*Default :* `false`
`Ineq`	BoolElt	*Default :* `false`
Verbose	`NormEquation`	*Maximum :* 1

Given quadratic field F and a non-negative integer m, return `true` if there exists an element α in the ring of integers O_F of F with norm m, and `false` otherwise. Instead of searching the maximal order O_F it is possible to search any suborder O of O_F for such element α by supplying O as a first argument.

For imaginary quadratic fields the method used is constructive (it uses Cornacchia's algorithm, see [Coh93] section 1.5.2), and if the value `true` is returned then a solution $[x]$ is also returned as a second return value.

Note that if the discriminant $F = \mathbf{Q}(\sqrt{d})$ with $d \equiv 1 \bmod 4$ (and squarefree) this function searches for a solution in integers to $x^2 + y^2 d = 4m$ (and the solution $\alpha = \frac{x+y\sqrt{d}}{2}$ is returned), whereas for $d \equiv 2, 3 \bmod 4$ a solution $\alpha = x + y\sqrt{d}$ with $x^2 + y^2 d = m$ in integers x, y is returned, if it exists. In an order of conductor f a search is conducted for a solution to the same equation with d replaced by $f^2 d$. Note that a version of `NormEquation` with integer arguments d and m also exists (see Section 18.12.2).

Unless m is the square of an integer, the factorization of m is used by the algorithm; if it is known, it may be supplied as the value of the optional parameter `Factorization` to speed up the calculation.

A verbose flag can be set to obtain some information on progress with the computation (see `SetVerbose` on page 1-102).

For real quadratic fields the same algorithm is used as for the general number fields. The last 4 parameters refer to this algorithm. See Section 34.7 for a description.

Example H35E5

```
> d := 302401481761723680;
> m := 7681481479118600246371 6;
> Q<z> := QuadraticField(-d);
> O<w> := sub< MaximalOrder(Q) | 6 >;
> f, s := NormEquation(O, m);
> s, Norm(s[1]);
406 + 1008*w 76814814791186002463716
```

35.4 Special Element Operations

There are a number of functions available only for elements of certain maximal orders. `Conjugate` is provided to return a quadratic element (instead of a real) as well as `ComplexConjugate`.

`a mod b`

The remainder on dividing a by b where a and b lie in the maximal order of $Q(\sqrt{d})$ for $d = -1, -2, -3, -7, -11, 2, 3, 5, 13$. `div` is provided for order elements in general, but for discriminants not in the above list `div` will fail if the division is not exact.

35.4.1 Greatest Common Divisors

`Gcd(a, b)`

`GCD(a, b)`

`GreatestCommonDivisor(a, b)`

The greatest common divisor of a and b in the maximal order of $\mathbf{Q}(\sqrt{d})$, where d must be one of the following values: $-1, -2, -3, -7, -11, 2, 3, 5, 13$.

`Lcm(a, b)`

`LCM(a, b)`

`LeastCommonMultiple(a, b)`

The least common multiple of a and b in the maximal order of $\mathbf{Q}(\sqrt{d})$, where d must be one of the following values: $-1, -2, -3, -7, -11, 2, 3, 5, 13$.

35.4.2 Modular Arithmetic

`Modexp(a, e, n)`

The computation of $a^e \bmod n$ in the maximal order of $\mathbf{Q}(\sqrt{d})$, where d is one of the following values: $-1, -2, -3, -7, -11, 2, 3, 5, 13$.

35.4.3 Factorization

MAGMA's factorization in maximal orders of quadratic number fields is based upon factoring the norm in the integers. Thus, the comments that are made about the `Factorization` command in the integers also apply here. Moreover, since the factorization may be off by a unit power, that power is also returned (the unit being -1, $\sqrt{-1}$, or $(1+\sqrt{-3})/2$).

```
Factorization(n)
Factorisation(n)
```

The factorization of n in the maximal order of the quadratic number field $Q(\sqrt{d})$, where d is one of: -1, -2, -3, -7, or -11. Returns the factorization along with the appropriate power of a unit (the unit being -1, $\sqrt{-1}$, or $(1+\sqrt{-3})/2$).

```
TrialDivision(n, B)
```

Trial division of n by primes of relative norm $\leq$ B in the maximal order of $Q(\sqrt{d})$, where d is one of: -1, -2, -3, -7, or -11. Returns the factored part, the unfactored part, and the power of the unit that the factorization is off by (the unit being -1, $\sqrt{-1}$, or $(1+\sqrt{-3})/2$).

35.4.4 Conjugates

```
ComplexConjugate(a)
```

The complex conjugate of quadratic field element a; returns a in a real quadratic field and $\bar{a} = x - y\sqrt{d}$ if $a = x + y\sqrt{d}$ in an imaginary quadratic field $\mathbf{Q}(\sqrt{d})$.

```
Conjugate(a)
```

The conjugate $x - y\sqrt{d}$ of $a = x + y\sqrt{d}$ in the quadratic field $\mathbf{Q}(\sqrt{d})$.

35.4.5 Other Element Functions

For the ring of integers of $\mathbf{Q}(\mathrm{i})$ the biquadratic residue symbol (generalizing the Legendre symbol) is available.

```
BiquadraticResidueSymbol(a, b)
```

Given a Gaussian integer a and a primary, non-unit Gaussian integer b, where a and b are coprime, return the value of the biquadratic character $\left(\frac{a}{b}\right)_4$. The value of this character is equal to i^k, for some $k \in \{0,1,2,3\}$. If a and b have a factor in common, the function returns 0, if b is not primary or b is a unit an error results.

```
Primary(a)
```

Return the unique associate $\bar{a}$ of the Gaussian integer a that satisfies

$$\bar{a} \equiv 1 \bmod (1+\mathrm{i})^3,$$

or 0 in case a is divisible by $1+\mathrm{i}$.

Example H35E6

The following example checks for primes p with $65 \le p \le 1000$ and $p \equiv 1 \bmod 4$ a result that was conjectured by Euler and proved by Gauss, namely that

$$z^4 \equiv 2 \bmod p \quad \text{has a solution} \iff p = x^2 + 64y^2 \quad \text{for some } x, y.$$

We use the function `NormEquation` to find the prime above p in the Gaussian integers, and we build the set of such primes for which 2 is a biquadratic residue (which means that $z^4 \equiv 2 \bmod p$ for some z).

```
> s := { };
> Q := QuadraticField(-1);
> M := RingOfIntegers(Q);
> for p := 65 to 1000 by 4 do
>    if IsPrime(p) then
>       _, x := NormEquation(Q, p);
>       if BiquadraticResidueSymbol(2, Primary(M!x[1])) eq 1 then
>          Include(~s, p);
>       end if;
>    end if;
> end for;
> s;
{ 73, 89, 113, 233, 257, 281, 337, 353, 577, 593, 601, 617, 881, 937 }
```

Next we create the set of all primes as above that are of the form $x^2 + 64y^2$. Note that we have to use `NormEquation` on a suborder of Q now, because we want to solve $x^2 + 64y^2 = p$, while `QuadraticField(-64)` returns just $\mathbf{Q}(i)$ in which we can only solve $x^2 + y^2 = p$.

```
> S := sub<MaximalOrder(Q) | 8>;
> t := { };
> for p := 65 to 1000 by 4 do
>    if IsPrime(p) then
>       if NormEquation(S, p) then
>           Include(~t, p);
>       end if;
>    end if;
> end for;
> t;
{ 73, 89, 113, 233, 257, 281, 337, 353, 577, 593, 601, 617, 881, 937 }
```

35.5 Special Functions for Ideals

Ideals of orders of quadratic fields inherit from ideals of orders of number fields (see Section 37.9 for the list of general functions and operations available). However there is also a correspondence between quadratic ideals and binary quadratic forms (see Chapter 33). The following functions make use of that correspondence.

`Content(I)`

The content of the ideal.

`Conjugate(I)`

The conjugate of the ideal.

`Discriminant(I)`

The discriminant of the quadratic form associated with I.

`QuadraticForm(I)`

The quadratic form associated with I.

`Ideal(f)`

The quadratic ideal with associated quadratic form f.

`Reduction(I)`

The quadratic ideal with associated quadratic form which is a reduction of the quadratic form associated to I.

35.6 Bibliography

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.

[**Jac99**] M. J. Jacobson, Jr. Applying sieving to the computation of quadratic class groups. *Math. Comp.*, 68(226):859–867, 1999.

36 CYCLOTOMIC FIELDS

Chapter 36
CYCLOTOMIC FIELDS

36.1 Introduction

Cyclotomic Fields (like the Quadratic Fields) are a subtype of the Number Fields (`FldNum`). They have some extra functionality which is described below and use some more efficient implementations. Orders of cyclotomic fields form the category `RngCyc` and the fields themselves `FldCyc`. Functions for cyclotomic fields and orders which work generally for number fields, their orders and elements are listed in Chapter 34.

There are two different representations of cyclotomic fields available:

* The "dense" representation: the field is conceptually represented as $Q(x)/f(x)$ where f is a cyclotomic polynomial, i.e., the minimal polynomial of a primitive root of unity.
* The "sparse" representation: Let $n = \prod p_i^{r_i}$ be the factorisation of n into prime powers and $n_i := p_i^{r_i}$. Then $Q(\zeta_n) = Q(\zeta_{n_1}, \ldots, \zeta_{n_r})$ and the field is represented as $Q(x_1, \ldots, x_r)/\langle f_{n_1}(x_1), \ldots, f_{n_r}(x_r)\rangle$.

As with the number fields, the non-simple representation, the issues are the same: the "sparse" representation allows for much larger fields – as long as the elements used have only few coefficients. The "dense" representation on the other hand has the asymptotically-fastest arithmetic.

36.2 Creation Functions

Functions are provided to create fields of the special type `FldCyc`. Orders and elements created from a field of this type will have the special types `RngCyc` and `FldCycElt` respectively and elements created from orders `RngCycElt`. These functions provide an object with the correct type which will allow the extra functions and efficient implementations to be used.

36.2.1 Creation of Cyclotomic Fields

Cyclotomic fields can be created from an integer specifying which roots of unity it should contain or from a collection of elements of an existing field or order. Cyclotomic polynomials can also be retrieved independently of the fields and orders.

```
CyclotomicField(m)
```

`Sparse`   Boolean   *Default :* `false`

Given a positive integer m, create the field obtained by adjoining the m-th roots of unity to **Q**. It is possible to assign a name to the primitive m-th root of unity ζ_m using angle brackets: `R<s> := CyclotomicField(m)`.

If `Sparse := true`, names for all the generating elements can be assigned.

`CyclotomicPolynomial(m)`

Given a positive integer m, create the cyclotomic polynomial of order m. This function is equivalent to `DefiningPolynomial(CyclotomicField(m))`.

`MinimalCyclotomicField(a)`

Given an element a from a cyclotomic field F or ring R, this function returns the smallest cyclotomic field or order thereof (possibly the rational field or the ring of integers) $E \subset F$ containing a.

`MinimalCyclotomicField(S)`

Given a set or sequence S of cyclotomic field or ring elements, this function returns the smallest cyclotomic field or ring (possibly the rational field or integers) G containing each of the elements of S.

Example H36E1

We will demonstrate the difference between the "dense" and the "sparse" representation on the cyclotomic field of order 100.

```
> K1 := CyclotomicField(100);
> K2 := CyclotomicField(100: Sparse := true);
> K2!K1.1;
zeta(100)_4*zeta(100)_25^19
```

Where `zeta(100)_25` indicates a 25th root of unity in a field of order 100.

```
> K1!K2.1;
zeta_100^25
```

36.2.2 Creation of Elements

For elements of cyclotomic number fields the following conventions are used. Primitive roots of unity ζ_m are chosen in such a way that $\zeta_m^{m/d} = \zeta_d$, for every divisor d of m; one may think of this as choosing $\zeta_m = \mathrm{e}^{\frac{2\pi\mathrm{i}}{m}}$ (where the roots of unity are $\zeta_m^k = \mathrm{e}^{\frac{2k\pi\mathrm{i}}{m}}$) in the complex plane for every m (a convention that is followed for the explicit embedding in the complex domains).

Elements of cyclotomic fields and orders can also be created using coercion (`!`) and the elt constructor (`elt<|>`) where the left hand side is the field or order the element will lie in. For details about coercion see Section 34.2.3.

`RootOfUnity(n)`

Create the n-th root of unity ζ_n in $\mathbf{Q}(\zeta_n)$.

`RootOfUnity(n, K)`

Given a cyclotomic field $K = \mathbf{Q}(\zeta_m)$ and an integer $n > 2$, create the n-th root of unity ζ_n in K. An error results if $\zeta_n \notin K$, that is, if n does not divide m (or $2m$ in case m is odd).

`Minimise(~a)`

`Minimize(~a)`

Given an element a in a cyclotomic field F or ring R, this procedure finds the minimal cyclotomic subfield $E \subset F$ or subring $E \subset F$ containing a, and coerces a into E. Note that E may be $\mathbf{Q}$ or $\mathbf{Z}$.

`Minimise(~s)`

`Minimize(~s)`

Given a set s of cyclotomic field or ring elements, this procedure finds the minimal cyclotomic field or ring E containing all of them, and coerces each element into E. The resulting set will have universe E. Note that E may be $\mathbf{Q}$ or $\mathbf{Z}$.

`Minimise(a)`

`Minimize(a)`

Given an element a in a cyclotomic field F or ring R, this function finds the minimal cyclotomic subfield $E \subset F$ or subring $E \subset R$ containing a, and coerces a into E. Note that E may be $\mathbf{Q}$ or $\mathbf{Z}$.

`Minimise(s)`

`Minimize(s)`

Given a set s of cyclotomic field or ring elements, this function finds the minimal cyclotomic field E containing all of them, and coerces each element into E. The resulting set will have universe E. Note that E may be $\mathbf{Q}$ or $\mathbf{Z}$.

36.3 Structure Operations

In cyclotomic fields the generic ring functions are supported (see Chapter 17). The functions listed below are those functions for cyclotomic fields which are additional to those for number fields. For the list of functions applying to general number fields see Section 34.2 and Section 34.3.

36.3.1 Invariants

`Conductor(K)`

The smallest n such that the field K is contained in $\mathbf{Q}(\zeta_n)$; for a cyclotomic field that is either the 'cyclotomic order' m (see below) or half that, depending on whether $m \equiv 2 \bmod 4$. The second return value is a sequence of the ramified real places of K.

`CyclotomicOrder(K)`

The value of m for the cyclotomic field $\mathbf{Q}(\zeta_m)$. Note that this will be the m with which the cyclotomic field was created.

`CyclotomicAutomorphismGroup(K)`

Returns the automorphism group of K as an abstract abelian group G and a map from G into the set of all automorphisms. Note that similar functionality is also available through `AutomorphismGroup` however, this function returns an abelian group and uses the fact that the automorphism group is already determined by the conductor.

`CyclotomicRelativeField(k, K)`

Given two cyclotomic fields $k \subseteq K$ a number field L/k is computed that is isomorphic to K.

36.4 Element Operations

For the full range of operations for elements of a number field or order see Section 34.4.

36.4.1 Predicates on Elements

Because of the nature of cyclotomic fields and orders, some properties of elements are easier to determine than in the general case.

`IsReal(a)`

Whether the cyclotomic field or ring element a is a real number, i.e., if it is invariant under the complex conjugation.

36.4.2 Conjugates

Elements of cyclotomic fields and orders can additionally have their complex conjugate computed. Conjugates are returned as cyclotomic elements (and not reals) and which conjugate is wanted can be indicated by providing a primitive root of unity.

`ComplexConjugate(a)`

The complex conjugate of cyclotomic field or ring element a.

```
Conjugate(a, n)
```

The image under the map $\zeta \mapsto \zeta^n$. The second argument (n) must be coprime to the conductor.

```
Conjugate(a, r)
```

The conjugate of the element $a \in \mathbf{Q}(\zeta_m)$ or its order, obtained by applying the field automorphism $\zeta_m \mapsto r$ where r is a primitive root of unity.

Example H36E2

The following lines of code generate a set W of minimal polynomials for the so-called Gaussian periods

$$\eta_d = \sum_{i=0}^{(l-1)/d-1} \zeta_l^{g^{di}}$$

where ζ_l is a primitive l-th root of unity (l is prime), and where g is a primitive root modulo l. These have the property that they generate a degree d cyclic subfield of $\mathbf{Q}(\zeta_l)$. We (arbitrarily) choose $l = 13$ in this example.

```
> R<x> := PolynomialRing(RationalField());
> W := { R | };
> l := 13;
> L<z> := CyclotomicField(l);
> M := Divisors(l-1);
> g := PrimitiveRoot(l);
> for m in M do
>    d := (l-1) div m;
>    g_d := g^d;
>    w := &+[z^g_d^i : i in [0..m-1] ];
>    Include(~W, MinimalPolynomial(w));
> end for;
```

Here is the same loop in just one line, using sequence reduction:

```
> W := { R | MinimalPolynomial(&+[z^(g^((l-1) div m))^i : i in [0..m-1] ]) :
>       m in M };
> W;
{
      x^12 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 +
       x + 1,
      x^6 + x^5 - 5*x^4 - 4*x^3 + 6*x^2 + 3*x - 1,
      x^3 + x^2 - 4*x + 1,
      x + 1
      x^4 + x^3 + 2*x^2 - 4*x + 3,
      x^2 + x - 3,
}
```

37 ORDERS AND ALGEBRAIC FIELDS

Chapter 37
ORDERS AND ALGEBRAIC FIELDS

37.1 Introduction

The number field module in MAGMA is based on the Kant/Kash system (Kant-V4) [KAN97], [KAN00], developed by the group of M. Pohst in Berlin.

The three main structures which this chapter is concerned with are `FldNum` (number fields), `RngOrd` (orders in number fields) and `FldOrd` (fields of fractions of `RngOrd`'s). Elements in these have types `FldNumElt`, `RngOrdElt` and `FldOrdElt` and ideals have types `RngOrdIdl` and `RngOrdFracIdl`. While number fields, and their sub-types cyclotomic fields and quadratic fields, are also detailed in separate chapters, (chapter 34 for number fields, 36 for cyclotomics and 35 for quadratics), almost all functionality described here applies to those objects as well.

On top of all there is a combined type that will match all "number field" types: `FldAlg` will match all references of type `FldNum`, `FldOrd`, `FldCyc`, and `FldQuad` (and similarly `FldAlgElt` will match all element types: `FldNumElt`, `FldOrdElt`, `FldCycElt` and `FldQuadElt`).

Number fields support extended types, they can be indexed by the type of the coefficient ring: `FldNum[FldRat]` refers to an absolute extension over $\mathbf{Q}$, while `FldNum[FldNum]` refers to a relative extension.

In order to use them efficiently, one has to understand the relations between the parent structures.

The basic distinction is between the number field point of view (`FldNum`) and the order based view (`FldOrd`, `RngOrd`).

We will start with the number fields. Formally, in MAGMA, an object K of type `FldNum` is an algebraic extension of finite degree over a a number field k or $\mathbf{Q}$. Thus it can be thought of as constructed as a quotient ring of a univariate polynomial ring over the base field modulo some irreducible polynomial: $K = k[t]/(f(t)k[t])$ Or, the field may be constructed as a multivariate quotient: $K = k[s_1, \ldots, s_n]/(f_1(s_1), \ldots, f_n(s_n))$ where all the polynomials are univariate. However, a slightly different representation is used internally.

It is important to remember that $\mathbf{Q}$ is **not** a number field.

One has to distinguish between number fields with primitive element α`:= K.1` which is a zero of f and number fields where no primitive element is known. In this case α_i`:= K.i` will be a zero of f_i.

However, number fields always have a 'power' basis, i.e. a basis containing only powers of the zero(s) of the defining polynomial(s) and products of those powers.

An important consequence of this representation as a quotient of a polynomial ring is that one cannot distinguish between e.g. $\mathbf{Q}[2^{(1/3)}]$ and $\mathbf{Q}[\zeta_3 2^{(1/3)}]$ – both of them are generated using a root of $t^3 - 2$. Therefore every non trivial extension generates a **new** object – even if the same polynomial is used repeatedly, except when the user explicitly tells MAGMA to check whether the polynomial has been used before.

An absolute extension is always an extension of **Q**. An arbitrary number field K can always be converted into an isomorphic extension of **Q** using a constructive variant of the primitive element theorem.

Likewise, if a subfield k of K is known, an isomorphic field as an extension of k can be computed.

The most important facts about the various representations are the following:

* Usually, arithmetic is fastest in absolute extensions. Thus, if one wants to do lots of basic arithmetic with the elements the transformation to an absolute representation is advisable. However, typically the operations are fastest when the elements are "small" in size.
* Invariants (like `Degree`, `Discriminant`, `Norm`, `Trace` etc.) are always relative to the current representation.
* Conversions of fields tend to be time consuming thus should be avoided if possible. However, once the different field representations are computed, the conversion of elements is not too time consuming.
* Some operations and invariants can (currently) only be done for absolute representations. Essentially, these are computations involving subfields and class and unit group computations.

Number fields support only arithmetic with their elements and the computation of some invariants (`GaloisGroup`, `Subfields`, `AutomorphismGroup`). Although invariants like the class group can be computed for `FldNums` this is only a shortcut for the corresponding computations for the maximal orders so e.g. `ClassGroup(K)` is expanded to `ClassGroup(MaximalOrder(K))`.

The other parent data-types are orders (`RngOrd`) and their fields of fractions (`FldOrd`). Orders can be constructed in basically two ways:

* as finite extensions E of a (maximal) order m (or of **Z**) by a zero of a monic integral polynomial $f \in m[t]$
* via a transformation from a different order.

The main restriction for the construction of orders is that the coefficient domain (`BaseRing`) must always be a maximal order if any structural computations are desired. An order O over some maximal order m is represented using a (pseudo) m-basis (of type `PMat` similar to the way modules over Dedekind rings (`ModDed`, `PMat`) are represented in general. Every other order must be free over its base ring.

An order O is equipped with a unique field of fractions, `FieldOfFractions(O)`, which has the same basis as the order and whose base field is the field of fractions of the base ring. The fields of fractions support almost no structural computations, they merely serve as a parent structure to any elements and ideals that (may) have denominators (i.e. that are non-integral w.r.t. the current structure).

From a practical point of view, orders and their field of fractions have two different element data types, namely elements (`RngOrdElt`) and ideals (`RngOrdIdl`) (resp. `FldOrdElt` and `RngOrdFracIdl`). This is technically not quite correct since ideals have formally different parents but those parents are trivial and the important information in them is the order the ideal is of. Ideals behave much more like elements than structures – they have no elements and are not rings in general. Formally, the parent structures obey the following rules:

* `MaximalOrder(BaseRing(K)) eq BaseRing(MaximalOrder(K))` for all number fields and fields of fractions K.

* `BaseField(FieldOfFractions(O)) eq FieldOfFractions(BaseRing(O))` for all orders O.

* `BaseField(NumberField(O)) eq NumberField(BaseRing(O))` for all orders O.

* all orders within the same field share the identical number field. For example: `IsIdentical(NumberField(O), NumberField(MaximalOrder(O)))`, while the corresponding fields of fractions will have, in general, different bases.

There are a few functions where orders behave differently to most other Magma objects, mostly because orders are not necessarily free modules over their base ring:

* `Parent(O.i) eq FieldOfFractions(O)` and furthermore `O.i in O` is usually `false`.

* `Eltseq` of an order element returns a sequence over the field of fractions of the base ring.

* The `O.i` are typically not a basis of the order – just part of a pseudo basis. However, they always form a basis of the field of fractions.

Algebraic number fields will be referred to as *number fields*. If a field may be either a number field or field of fractions it will be referred to as an *algebraic field*.

37.2 Creation Functions

The following describes how number fields, orders and their fields of fractions may be created. It also shows some ways of creating elements of these rings and homomorphisms from these rings into an arbitrary ring.

37.2.1 Creation of General Algebraic Fields

Algebraic Number Fields can be created in a various ways, all involving polynomials. The fields can be created as absolute extensions, i.e. an extension of **Q** by an irreducible polynomial, or as a relative extension which is an extension of an algebraic field by a polynomial irreducible over that field. Fields of fractions can be created as `FieldOfFractions(O)` where `O` is some order or from an existing field of fractions by using some of the constructors and functions described below. See also the corresponding sections for number fields section 34.2.1, section 36.2 for cyclotomics and section 35.2 for quadratics.

`NumberField(f)`

`Check`	BoolElt	*Default :* `true`
`DoLinearExtension`	BoolElt	*Default :* `false`

`Global`	BoolElt	*Default :* `false`

Given an irreducible polynomial f of degree $n \geq 1$ over $K = \mathbf{Q}$ or some number field K, create the number field $L = K(\alpha)$ obtained by adjoining a root α of f to K. For details see `NumberField` in Section 34.2.1.

The angle bracket notation may be used to assign the root α to an identifier e.g. `L<y> := NumberField(f)` where y will be a root of f.

```
RationalsAsNumberField()
QNF()
```

This creates a number field isomorphic to $\mathbf{Q}$. It is equivalent to `NumberField(x-1) : DoLinearExtension)`, where x is `PolynomialRing(Rationals()).1`.

The result is a field isomorphic to $\mathbf{Q}$, but regarded by Magma as a number field (while $\mathbf{Q}$ itself is not, since `FldRat` is not a subtype of `FldNum`). It therefore supports all of the number field functions, while the `Rationals()` do not. On the other hand, arithmetic will be slower.

Coercion can be used to convert to and from the `Rationals()`.

```
NumberField(s)
```

`Check`	BoolElt	*Default :* `true`
`DoLinearExtension`	BoolElt	*Default :* `false`
`Abs`	BoolElt	*Default :* `false`

Let K be a possibly trivial algebraic extension of $\mathbf{Q}$. Given a sequence s of nonconstant polynomials $s_1, \ldots, s_m$, that are irreducible over K, create the number field $L = K(\alpha_1, \ldots, \alpha_m)$ obtained by adjoining a root α_i of each s_i to K. For details see `NumberField` in Section 34.2.1.

```
ext<  F | s1, ..., sn  >
ext< F | s >
```

`Check`	BoolElt	*Default :* `true`
`Global`	BoolElt	*Default :* `false`
`Abs`	BoolElt	*Default :* `false`
`DoLinearExtension`	BoolElt	*Default :* `false`

Create the algebraic field defined by extending F by the polynomials s_i or the polynomials in the sequence s. Similar as for `NumberField(S)` described above, F may be $\mathbf{Q}$ or a field of fractions. If F is a field of fractions a field of fractions will be returned otherwise a number field will be returned. A tower of fields similar to that of `NumberField` is created and the same restrictions as for that function apply to the polynomials that can be used in the constructor.

`RadicalExtension(F, d, a)`

`Check`	BoolElt	*Default :* `true`

Let F be an algebraic field. Let a be an integral element of F chosen such that a is not an n-th power for any n dividing d. Returns the algebraic field obtained by adjoining the d-th root of a to F.

`SplittingField(F)`

`Abs`	BoolElt	*Default :* `true`
`Opt`	BoolElt	*Default :* `true`

Given an algebraic field F, return the splitting field of its defining polynomial. The roots of the defining polynomial in the splitting field are also returned.

If `Abs` is `true`, the resulting field will be an absolute extension, otherwise a tower is returned.

If `Opt` is `true`, an attempt of using `OptimizedRepresentation` is done. If successful, the resulting field will have a much nicer representation. On the other hand, computing the intermediate maximal orders can be extremely time consuming.

`SplittingField(f)`

Given an irreducible polynomial f over $\mathbf{Z}$, return its splitting field.

`SplittingField(L)`

`Abs`	BoolElt	*Default :* `false`
`Opt`	BoolElt	*Default :* `false`

Given a sequence L of polynomials over a number field or the rational numbers, compute a common splitting field, i.e. a field K such that every polynomial in L splits into linear factors over K. The roots of the polynomials are returned as the second return value.

If the optional parameter `Abs` is `true`, then a primitive element for the splitting field is computed and the field returned will be generated by this primitive element over $\mathbf{Q}$. If in addition `Opt` is also `true`, then an optimized representation of K is computed as well.

`sub< F | e`$_1$`, ..., e`$_n$` >`

Given an algebraic field F with ground field G and n elements $e_i \in F$, return the algebraic field $H = G(e_1, \ldots, e_n)$ generated by the e_i (over G), as well as the embedding homomorphism from H to F.

```
MergeFields(F, L)
```
```
CompositeFields(F, L)
```

Let F and L be absolute algebraic fields. Returns a sequence of fields $[M_1, \ldots, M_r]$ such that each field M_i contains both a root of the generating polynomial of F and a root of the generating polynomial of L.

In detail: Suppose that F is the smaller field (wrt. the degree). As a first step we factorise the defining polynomial of L over F. For each factor obtained, an extension of F is constructed and then transformed into an absolute extension. The sequence of extension fields is returned to the user.

```
Compositum(K, L)
```

For absolute number fields K and L, at least one of which must be normal, find a smallest common over field. Note that in contrast to `CompositeFields` above the result here is essentially unique since one field was normal.

```
Compositum(K, A)
```

For an normal number field K and abelian extension A of some subfield of K, find a smallest common over field. Note that in contrast to `CompositeFields` above the result here is essentially unique since K is normal.

```
OptimizedRepresentation(F)
```
```
OptimisedRepresentation(F)
```
```
OptimizedRepresentation(F, d)
```
```
OptimisedRepresentation(F, d)
```

Given an algebraic field F with ground field $\mathbf{Q}$, this function will attempt to find an isomorphic field L with a better defining polynomial (in the sense defined below) than the one used to define F. If such a polynomial is found then L is returned; otherwise F will be returned.

If the argument d is not specified, a polynomial g with integer coefficients is defined to be better than f if it is monic, irreducible, defines a number field isomorphic to F and its discriminant is smaller (in absolute value) than that of f. If a second argument d is specified, then g is defined to be better if in addition to the previous requirements d is not an index divisor, that is, if d does not divide the index (defined in the Invariants sub–section) $[O_L : E_L]$ of the equation order E_L of L in the maximal order O_L, (which are defined in the next sub–section).

Note however, that as a first step this function will determine the maximal order of F which may take some time if the field is large.

Example H37E1

Some results of `OptimizedRepresentation` are shown.

```
> R<x> := PolynomialRing(Rationals());
> K := NumberField(x^4-420*x^2+40000);
> L := OptimizedRepresentation(K);
> L ne K;
true
> L;
Number Field with defining polynomial x^4 - 4*x^3 -
    17*x^2 + 42*x + 59 over the Rational Field
> L eq OptimizedRepresentation(L);
> f := DefiningPolynomial(L);
> Z := IntegerRing();
> Factorization(Z ! Discriminant(f));
[ <2, 18>, <5, 8>, <41, 2> ]
> g := x^4 - 4*x^3 - 17*x^2 + 42*x + 59;
> Factorization(Z ! Discriminant(g));
[ <2, 4>, <3, 4>, <5, 2>, <41, 2> ]
> OL := MaximalOrder(L);
> EL := EquationOrder(L);
> Index(OL, EL);
36
> OptimizedRepresentation(L, 2) eq L;
true
```

As we see from this computation, the prime 5 (as well as 41) divides the discriminant of g twice. This means that, potentially, 5 would still divide the index of the equation order in the maximal order O_L of L. However, in fact E_L has only index 36 in O_L.
The optimized representation of L such that 2 does not divide the index of E_L in O_L is L so 2 does divide the index seemingly contrary to the description above. However, if a more optimal representation cannot be found then the field is returned which is what happens here.

37.2.2 Creation of Orders and Fields from Orders

The maximal order $\mathcal{O}_K$ of an algebraic field and the equation order of a number field can be obtained from the field. Other orders of a field are unitary subrings of finite index in the ring of integers; they contain a subset of the integral elements in the field. The equation order $\mathcal{E}_K = \mathbf{Z}[\alpha]$ of $K = \mathbf{Q}(\alpha) \cong \mathbf{Q}[X]/f(X)$, where K is a number field defined by a monic integral polynomial, has the same basis as K, a power basis. Obviously $\mathcal{E}_K \subset \mathcal{O}_K$ since the minimal polynomial of α is integral and monic. Once an order is created in Magma further orders can be created from it.

`EquationOrder(f)`

Check BoolElt *Default :* `true`

Given an irreducible non-constant monic integral polynomial $f \in R[X]$, return the equation order $E = \mathbf{R}[X]/f(X)$ corresponding to f. If the optional parameter `Check` is set to `false` then the polynomial will not be checked for irreducibility.

`EquationOrder(K)`

Return the equation order corresponding to the polynomial with which the number field K was defined. K must have been defined by a monic integral polynomial. Thus this function returns the extension of the equation order of the ground field of K by the defining polynomial of K.

`SubOrder(O)`

Provided the order O is not an equation order, O is a transformation of some order O'. This function returns O'.

`EquationOrder(O)`

A suborder of the order O which is defined by a polynomial. e.g. $R[x]/f$ where R is a polynomial ring over the coefficient ring of O and f is in R. It will also be the final order of `SubOrder(SubOrder(... SubOrder(O)))`. O must have a monic defining polynomial for the equation order to exist.

`Integers(O)`

`RingOfIntegers(O)`

`IntegerRing(O)`

Returns the ring of integers in the order O, ie. O itself.

Example H37E2

Once a number field $K = \mathbf{Q}(\alpha)$ has been created, one can obtain the equation order $E = \mathbf{Z}[\alpha]$ and the ring of integers O_K simply as follows.

```
> R<x> := PolynomialRing(Integers());
> K := NumberField(x^4-420*x^2+40000);
> E := EquationOrder(K);
> O := MaximalOrder(K);
> Index(O, E);
64000
```

Note that entirely different things happen here: for the equation order nothing has to be computed, but the determination of the maximal order involves the complicated Round 2 (or 4) algorithm. In our particular example above, E is a subring of index $2^9 \cdot 5^3$ in O (see also the example for orders and ideals in the subsection 37.2.3.1 where a maximal order is created).

```
sub< O | a_1, ..., a_r >
```

Create the suborder of the order O generated (as an algebra over $\mathbf{Z}$) by the elements $a_1, \ldots, a_r \in O$, that is, create $\mathbf{Z}[a_1, \ldots, a_r]$. If the algebra does not have full rank as a sub-module of O, an error results. Note, however, that it is currently *not* required that 1 is in the sub-ring.

```
ext< O | a_1, ..., a_r >
```

Given an order O, and elements $a_1, \ldots, a_r$ lying in the maximal order of O, create the order $O[a_1, \ldots, a_r]$. Note that using this constructor O can only be extended to be as large as the maximal order. This does not cause the maximal order to get computed. See also `Order` for a different version that allows parameters to improve efficiency.

```
ext< O | f >
```

Given an order O and a polynomial f of degree n with coefficients in O, create the extension E of O by a root of f which forms a free module of rank n over O : $E \cong O[\alpha]$; it is necessary for f to be irreducible over O.

```
FieldOfFractions(O)
```

Return the field containing all fractions of elements of O. The angle bracket notation can be used to assign names to the basis elements of F and assign these elements to variables, e.g. `F<x, y> := FieldOfFractions(MaximalOrder(x^2 + 3))`.

```
Order(F)
```

The order of which F was created as its field of fractions. This function is an inverse to `FieldOfFractions`.

```
NumberField(O)
```

The number field of an order is recursively defined by:

1. the number field of $\mathbf{Z}$ is $\mathbf{Q}$
2. the number field of O is the number field of the coefficient ring of O, (i.e. the order over which O is defined), with an element α adjoined where α is a root of the defining polynomial of O.

```
NumberField(F)
```

The number field of `Order(F)` for a field of fractions F.

Example H37E3

The following illustrates the relationship between the bases of an order, its field of fractions and its number field.

```
> R<x> := PolynomialRing(Integers());
> f := x^5 + 5*x^4 - 75*x^3 + 250*x^2 + 65625;
> M := MaximalOrder(f);
> M;
Maximal Order of Equation Order with defining polynomial x^5 + 5*x^4 - 75*x^3 +
    250*x^2 + 65625 over its ground order
> Basis(FieldOfFractions(M));
[
    M.1,
    M.2,
    M.3,
    M.4,
    M.5
]
> Basis(NumberField(M));
[
    1,
    $.1,
    $.1^2,
    $.1^3,
    $.1^4
]
> Basis(M);
[
    M.1,
    M.2,
    M.3,
    M.4,
    M.5
]
> M.1 eq 1;
true
> M.2 eq NumberField(M).1;
false
> E := EquationOrder(M);
> NumberField(M) eq NumberField(E);
true
> Basis(FieldOfFractions(E), NumberField(M));
[
    1,
    $.1,
    $.1^2,
    $.1^3,
    $.1^4
]
> M!Basis(FieldOfFractions(E))[1];
[1, 0, 0, 0, 0]
> M!Basis(FieldOfFractions(E))[2];
[0, 5, 0, 0, 0]
> M!NumberField(M).1;
[0, 5, 0, 0, 0]
```

`OptimizedRepresentation(O)`

`OptimisedRepresentation(O)`

`OptimizedRepresentation(O, d)`

`OptimisedRepresentation(O, d)`

Given an order O with ground ring $\mathbf{Z}$, this function will attempt to find an isomorphic maximal order M with a better defining polynomial than the one used to define O. If such a polynomial is found then M is returned; otherwise O will be returned.

If the argument d is not specified, a polynomial g with integer coefficients is defined to be better than f if it is monic, irreducible, defines an order isomorphic to O and its discriminant is smaller in absolute value than that of f. If a second argument d is specified, then g is defined to be better if in addition to the previous requirements d is not an index divisor, that is, if d does not divide the index (defined in the Invariants sub–section) $[M : E_M]$ of the equation order E_M of M in M.

`O + P`

Add two orders O and P having the same equation order. Computes the smallest common over order.

`O meet P`

The intersection of two orders O and P having the same equation order.

`AsExtensionOf(O, P)`

Return the order O as an transformation of the order P where O and P have the same coefficient ring.

`Order(O, T, d)`

`Check` BoolElt *Default :* `true`

Let O be an absolute order with basis $b_1, \ldots, n_n$, $T = (T_{i,j}) \in \mathrm{GL}(n, \mathbf{Q}) \cap \mathrm{Mat}(n, \mathbf{Z})$ and $d \in \mathrm{N}$. This function creates the order with basis $(1/d \sum_{j=1}^{n} T_{i,j} b_j)_{i \le i \le n}$. `Check` can be set to `false` when the order is large to avoid checking that the result actually is an order.

```
Order(O, M)
```

Let O be an order with (pseudo) basis $b_1, \ldots, b_n$ and $M = \sum_{i=1}^n A_i\alpha_i \subseteq k^n$ be an o_k-module where o_k is the coefficient ring of O. This function creates the order $\sum_{i=1}^n A_i c_i$ where $c_i := \sum_{j=1}^n \alpha_{i,j} b_j$. If `Check` is set to `false`, then it will not be checked that the result actually is an order (potentially expensive).

```
Order( [ e1, ...  en ] )
```

`Verify`	BoolElt	*Default :* `true`
`Order`	BoolElt	*Default :* `false`

Given n elements $e_1, \ldots, e_n$ in an algebraic extension field F over $\mathbf{Q}$ create the minimal order O of F which contains all the e_i. If `Verify` is `true`, it is verified that the e_i are integral algebraic numbers. This can be a lengthy process if the field is of large degree.

Setting `Order` to `true` assumes that the given elements actually form a basis for the new order, thus it avoids testing for multiplicative closure. Without this parameter the order returned will have a canonical basis chosen with no direct relation to the input. By default, products of the generators will be added until the module is closed under multiplication.

37.2.3 Maximal Orders

The maximal order $\mathcal{O}_K$ is the ring of integers of an algebraic field consisting of all integral elements of the field; that is, elements which are roots of monic integer polynomials. It may also be called the number ring of a number field.

There are a number of algorithms which Magma uses whilst computing maximal orders. Each maximal order is a sum of p-maximal orders. The main algorithm used for p-maximal orders is a mixture of the Round–2 and Round–4 methods ([Coh93, Bai96, Poh93, PZ89]) for absolute extensions and a variant of the Round–2 for relative extensions ([Coh00, Fri97]).

However if the field is a radical (pure) extension, there is another algorithm available which is used to calculate each p-maximal order. In this case we can compute a pseudo basis for the p-maximal orders knowing only the valuation of the constant coefficient of the defining polynomial at p [Sut12].

The Round–2 and Round–4 algorithms can be selected by setting the parameter `Al` to `"Round2"` and `"Round4"` respectively. Another option for this parameter and for computation of p-maximal orders is the `"Pauli"` method. This method is only available for equation orders in simple relative extensions. It uses the factorization of the defining polynomial over the completion of the order.

Alternatively, if the discriminant of the maximal order is already known, the parameters `Discriminant` or `Ramification` can be used. If the input is an order O and the `Discriminant` or `Ramification` parameters are supplied an algorithm which can compute the maximal order given the discriminant of the maximal order will be used. `Discriminant` must be an integer if O is an absolute order and must be an ideal of the coefficient ring of O if O is a relative order. `Ramification` must contain integers if O is an absolute order and must contain ideals of the coefficient ring of O if O is a relative order. The ramification sequence is taken to contain prime factors of the discriminant. Only one of these parameters can be specified and if one of them is then `Al` cannot be specified. This algorithm is based on [Bj94], Theorems 1.2 and 7.6.

`MaximalOrder(O)`

`Al`	MonStgElt	*Default :* "*Auto*"

`MaximalOrder(F)`

`IntegerRing(F)`

`Integers(F)`

`RingOfIntegers(F)`

`Discriminant`	Any	*Default :*
`Ramification`	SeqEnum	*Default :*
Verbose	`MaximalOrder`	*Maximum :* 5

Return the maximal order or ring of integers of the number field F. When the input is an order O or a field of fractions of an order O return the order containing O which is the largest order in the number field of O.

An integral basis for F can be found as the basis of the maximal order.

For information on the parameters, see the introduction to this section above, 37.2.3.

`MaximalOrder(f)`

`Check`	BoolElt	*Default :* `true`
`Al`	MonStgElt	*Default :* "*Auto*"
`Discriminant`	Any	*Default :*
`Ramification`	SeqEnum	*Default :*
Verbose	`MaximalOrder`	*Maximum :* 5

This is equivalent to `MaximalOrder(NumberField(f))`.

The `Check` parameter if set to `false` will prevent checking of the polynomial for irreducibility.

For information on the other parameters, see the introduction to this section above.

Example H37E4

The following shows the advantage of the `Ramification` parameter to the `MaximalOrder` function.

```
> R<t> := PolynomialRing(Integers());
> f1 := t^14 - 63*t^12 - 9555*t^11 + 118671*t^10 - 708246*t^9 - 17922660*t^8 +
> 859373823*t^7 + 2085856500*t^6 - 117366985106*t^5 - 335941176396*t^4 +
> 4638317668005*t^3 + 17926524826973*t^2 + 7429846568445*t+ 91264986397629;
> d1 := [2, 3, 5, 7, 59];
> time MaximalOrder(f1:Ramification := d1);
Maximal Order of Equation Order with defining polynomial x^14 - 63*x^12 -
    9555*x^11 + 118671*x^10 - 708246*x^9 - 17922660*x^8 + 859373823*x^7 +
    2085856500*x^6 - 117366985106*x^5 - 335941176396*x^4 + 4638317668005*x^3 +
    17926524826973*x^2 + 7429846568445*x + 91264986397629 over Z
Time: 0.230
> time MaximalOrder(f1);
Maximal Order of Equation Order with defining polynomial x^14 - 63*x^12 -
    9555*x^11 + 118671*x^10 - 708246*x^9 - 17922660*x^8 + 859373823*x^7 +
    2085856500*x^6 - 117366985106*x^5 - 335941176396*x^4 + 4638317668005*x^3 +
    17926524826973*x^2 + 7429846568445*x + 91264986397629 over Z
Time: 0.590
> f2 := t^14 - 129864*t^12 - 517832*t^11 + 6567239322*t^10 + 33352434192*t^9 -
> 166594899026864*t^8 - 752915315481312*t^7 + 2275891736459084940*t^6 +
> 7743078094604088768*t^5 - 16633213695413438344032*t^4 -
> 39871919309692447523616*t^3 + 60126791399546070679893112*t^2 +
> 77844118533852728698751040*t - 83173498199506854751458701376;
> d2 := [2,3,7,4145023];
>
> time MaximalOrder(f2:Ramification := d2);
Maximal Order of Equation Order with defining polynomial x^14 - 129864*x^12 -
    517832*x^11 + 6567239322*x^10 + 33352434192*x^9 - 166594899026864*x^8 -
    752915315481312*x^7 + 2275891736459084940*x^6 + 7743078094604088768*x^5 -
    16633213695413438344032*x^4 - 39871919309692447523616*x^3 +
    60126791399546070679893112*x^2 + 77844118533852728698751040*x -
    83173498199506854751458701376 over Z
Time: 0.730
> time MaximalOrder(f2);
Maximal Order of Equation Order with defining polynomial x^14 - 129864*x^12 -
    517832*x^11 + 6567239322*x^10 + 33352434192*x^9 - 166594899026864*x^8 -
    752915315481312*x^7 + 2275891736459084940*x^6 + 7743078094604088768*x^5 -
    16633213695413438344032*x^4 - 39871919309692447523616*x^3 +
    60126791399546070679893112*x^2 + 77844118533852728698751040*x -
    83173498199506854751458701376 over Z
Time: 0.840
> f13 := t^15 - 114*t^14 + 282185319*t^13 + 1247857228852*t^12 -
> 35114805704965233*t^11 - 141524337796433387826*t^10 +
> 2604584980442264028744009*t^9 + 14153948932132918272984150384*t^8 -
> 178273077248353369941327628479552*t^7 - 1142953506821390914419260564494304768
> *t^6 + 15975069142211276963134599495014990639616*t^5 +
> 33516684438303088018217308253251277376159744*t^4 -
> 617589777108203716232396372453619309554471256064*t^3 -
> 3975614124450669195454617623548846315018061748633360*t^2 +
> 22666571829085475706482454642151923578020471016281866624*t
> - 130222245653276025640691622325930696056165742877781419622 4;
> d13 := [2, 3, 3377890562461, 7623585272461];
> time MaximalOrder(f13: Ramification := d13);
Maximal Order of Equation Order with defining polynomial x^15 - 114*x^14 +
    282185319*x^13 + 1247857228852*x^12 - 35114805704965233*x^11 -
    141524337796433387826*x^10 + 260458498044226402874 4009*x^9 +
    1415394893213291827298415 0384*x^8 - 17827307724835336994132762847 9552*x^7 -
    11429535068213909144192605644943 04768*x^6 +
    159750691422112769631345994950149906 39616*x^5 +
    3351668443830308801821730825325127737 6159744*x^4 -
    6175897771082037162323963724536193095544712 56064*x^3 -
    39756141244506691954546176235488463150180617486 3360*x^2 +
    22666571829085475706482454642151923578020471016281 86624*x -
    13022224565327602564069162232593069605616574287778141 96224 over Z
Time: 0.270
>  time MaximalOrder(f13);
```

The last line did not complete running in over 7 hours.

37.2.3.1 Orders and Ideals

Orders may be created using ideals of another order. Ideals are discussed in Section 37.9. The following intrinsics form part of the computation of maximal orders as discussed above in Section 37.2.3.

`pMaximalOrder(O, p)`

`Al` MonStgElt *Default : "Auto"*

The p-maximal overorder of O (see also the example below). This is the largest overorder P such that the index $(P : O)$ is a power of p, a prime in the coefficient ring of O. The options for the `Al` parameter are the same as those for `MaximalOrder`.

If O is a kummer extension then specific code is used to calculate each p-maximal order, rather than the Round 2 or Round 4 methods. In this case we know 1 or 2 elements which generate the p-maximal order and can easily write the order down.

`pRadical(O, p)`

Returns the p-radical of an order O for a prime p in the coefficient ring of O, defined as the ideal consisting of elements of O for which some power lies in the ideal pO.

It is possible to call this function for p not prime (so long as p is greater than the degree of O if it is an integer). In this case the p-trace-radical will be computed, i.e.

$$\{x \in F \mid \mathrm{Tr}(xO) \subseteq p\mathbf{Z}\}.$$

If p is square free and all divisors are larger than the field degree, this is the intersection of the radicals for all l dividing p. In particular together with `MultiplicatorRing` this can sometime be used to compute maximal orders without factoring the discriminant [Bj94, Fri00] or at least "good" approximations.

`MultiplicatorRing(I)`

Returns the multiplicator ring M of the ideal I of the order O, that is, the subring of elements of the field of fractions K of O multiplying I into itself: $M = \{x \in F : xI \subset I\}$.

Example H37E5

To illustrate how the Round 2 algorithm for the determination of the ring of integers works, we present an implementation of it in the Magma language. The key functions are `MultiplicatorRing` and `pRadical`, called by the following function `pMaximalOverOrder`;

```
> pMaximalOverOrder := function(ord, p)
>       ovr := MultiplicatorRing(pRadical(ord, p));
>       print "index is", Index(ovr, ord);
>       return (Index(ovr, ord) eq 1) select ovr else $$(ovr, p);
> end function;
```

which finds the largest overorder in which the given order has p-power index. This function is now simply applied to the equation order, for each prime dividing the discriminant:

```
> Round2 := function(E, K)
>       // E should be some order of a number field K
>       d := Discriminant(E);
>       fact := Factorization(Abs(d));
>       print fact;
>       M := E;
>       for x in fact do
>            M := M+pMaximalOverOrder(E, x[1]);
>       end for;
>       print "index of equation order in maximal order is:", Index(M, E);
>       return M;
> end function;
```

In our running example, this produces the following output:

```
> R<x> := PolynomialRing(Integers());
> K := NumberField(x^4-420*x^2+40000);
> E := EquationOrder(K);
> Round2(E, K);
[ <2, 18>, <5, 8>, <41, 2> ]
index is 2
index is 4
index is 8
index is 4
index is 2
index is 1
index is 5
index is 25
index is 1
index is 1
index of equation order in maximal order is: 64000
Transformation of E
Transformation Matrix:
[800   0   0   0]
[  0 400   0   0]
[  0 200  20   0]
```

```
[400 180   0   1]
Denominator: 800
```

37.2.4 Creation of Elements

Elements of algebraic fields and of orders are displayed quite differently. Algebraic field elements are always printed as a linear combination with rational coefficients of the basis elements of the field. For number fields which have a power basis this is also a polynomial in the primitive element of the field with rational coefficients; that is, an element of $G[x]/f$, where f was the defining polynomial of the field over the ground field G. Since in general G will be an algebraic field itself, elements in relative extensions, i.e. G strictly bigger than $\mathbf{Q}$, will be printed as linear combinations with linear combinations as coefficients and for number fields this will look like multivariate polynomials. In fact they are recursively defined univariate polynomials.

Elements of orders are displayed as sequences of integer coefficients, referring to the basis of the order. To convert this $\mathbf{Z}$-basis representation to a polynomial expression in the primitive element of an associated number field, the element should be coerced into the number field (using `!`). To print the element as a linear combination of the basis elements, coerce the element into the field of fractions.

`F ! a`

`elt< F | a >`

Coerce a into the field F. Here a may be an integer or a rational field element, or an element from a subfield of F, or from an order in such.

`F ! [a`$_0$`, a`$_1$`, ..., a`$_{m-1}$`]`

`elt< F | [ a`$_0$`, a`$_1$`, ..., a`$_{m-1}$` ] >`

`elt< F | a`$_0$`, a`$_1$`, ..., a`$_{m-1}$` >`

Given the algebraic field, F of degree m over its ground field G and a sequence $[a_0, \ldots, a_{m-1}]$ of elements of G, construct the element $a_0\alpha_0 + a_1\alpha_1 + \cdots a_{m-1}\alpha_{m-1}$ of F where the α_i are the basis elements of F.

`O ! a`

`elt< O | a >`

Coerce a into the order O. Here a is allowed to be an integer, or an integral element of an associated algebraic field of O, or an element of a quotient order.

`O ! [a`$_0$`, a`$_1$`, ..., a`$_{m-1}$`]`

`elt< O | [ a`$_0$`, a`$_1$`, ..., a`$_{m-1}$` ] >`

`elt< O | a`$_0$`, a`$_1$`, ..., a`$_{m-1}$` >`

Given the order O of degree m and elements $a_0, a_1, \ldots, a_{m-1}$ in the ground order of O, construct the element $a_0\alpha_0 + a_1\alpha_1 + \cdots + a_{m-1}\alpha_{m-1}$ of O, where $\alpha_0, \ldots, \alpha_{m-1}$ is the basis for the order.

`Random(F, m)`

`Random(O, m)`

A random element of the algebraic field F or order O. The maximal size of the coefficients is determined by m.

`Random(I, m)`

A random element of the ideal I as an element of the field of fractions of the associated order. The maximal size of the coefficients with respect to the ideal basis is determined by m.

Example H37E6

Here are three ways of creating the same integral element in K as an element of the maximal order and its field of fractions.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^4-420*x^2+40000);
> O := MaximalOrder(K);
> e := O ! (y^2/40 + y/4);
> f := elt< O | [0, 0, 1, 0]>;
> f eq e;
true
> F<a, b, c, d> := FieldOfFractions(O);
> g := F![0, 0, 1, 0];
> g eq e;
true
> g;
c
```

These constructions would have failed if the element was not in O.

`One(K)` `One(O)` `Identity(K)` `Identity(O)`

`Zero(K)` `Zero(O)` `Representative(K)` `Representative(O)`

37.2.5 Creation of Homomorphisms

To specify homomorphisms from algebraic fields or orders in algebraic fields, it is necessary to specify the image of the generating elements, and possible to specify a map on the ground field.

```
hom< F -> R | r >
```

```
hom< F -> R | h, r >
```

Given an algebraic field F, defined as an extension of the ground field G, as well as some ring R, build the homomorphism ϕ obtained by sending the defining primitive element α of F to the element $r \in R$.

If F is a field of fractions then r will be the image of the primitive element of the field of fractions of the equation order of `Order(F)`.

It is possible (if $G = \mathbf{Q}$) and sometimes necessary (if $G \neq \mathbf{Q}$) to specify a homomorphism ϕ on F by specifying its action on G by providing a homomorphism h with G as its domain and R its codomain together with the image of α. If R does not cover G then the homomorphism h from G into R is necessary to ensure that the ground field can be mapped into R.

```
hom< O -> R | r >
```

```
hom< O -> R | h, r >
```

Given an order O, a ring R and an element $r \in R$, construct a homomorphism ϕ by sending the primitive element of the equation order of O to r.

To be more precise: Let K be the field of fractions of O, k be the base field of K i.e. the field of fractions of the base ring of O, and $N :=$ `NumberField(O)`. As a k–algebra K is generated by $x := K!N.1$, so x is zero of `DefiningPolynomial(O)`. The element r given in the map construction will be the image of x.

When O is an equation order e.g. if O was defined using a monic integral polynomial x will be `O.2`.

As in the field case it is possible to specify a map on the coefficient ring of O (ring over which O is defined) with codomain R. This is necessary if R does not cover the coefficient ring of O.

Example H37E7

We show a way to embed the field $\mathbf{Q}(\sqrt{2})$ in $\mathbf{Q}(\sqrt{2}+\sqrt{3})$. The application of the homomorphism suggests how the image could have been chosen.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^2-2);
> KL<w> := NumberField(x^4-10*x^2+1);
> H := hom< K -> KL | (9*w-w^3)/2 >;
> H(y);
1/2*(-w^3 + 9*w)
> H(y)^2;
```

```
2
```

Homomorphisms can be created between any order or algebraic field and any ring.

```
> f := x^4 + 5*x^3 - 25*x^2 + 125*x + 625;
> M := MaximalOrder(f);
> F<a, b, c, d> := FieldOfFractions(M);
> FF := FiniteField(5, 3);
> F;
Field of Fractions of M
> FF;
Finite field of size 5^3
> h := hom< F -> FF | Coercion(Rationals(), FF), 3*FF.1>;
> h;
Mapping from: FldOrd: F to FldFin: FF
>  h(a); h(b);
1
>> h(a); h(b);
          ^
Runtime error in map application: Application of map failed
> h(5*b); h(5*5*c); h(5*5*5*d);
FF.1^94
FF.1^64
FF.1^34
```

This unexpected behaviour occurs because when the basis of F is expressed with respect to the power basis of the number field they have denominator divisible by 5. A more well–behaved example is shown below.

```
> FF := FiniteField(11, 5);
> h := hom< F -> FF | Coercion(Rationals(), FF), 7*FF.1>;
> h(a);
1
> h(b); h(c); h(d);
FF.1^48316
FF.1^96632
FF.1^144948
> 7*FF.1;
FF.1^112736
> 5*h(b);
FF.1^112736
> PrimitiveElement(F);
5/1*b
```

`hom< O -> R |` $b_1, ..., b_n$ `>`

`hom< O -> R | m,` $b_1, ..., b_n$ `>`

Return the map from the order O of an algebraic number field to the ring R mapping the basis elements to $b_1, .., b_n$. If given, the map m should be from the coefficient ring of O to R and will be used to map the coefficients of the basis elements. If not given the coefficient ring of O should by covered by R.

`IsRingHomomorphism(m)`

Return whether the vector space homomorphism m is a homomorphism of rings.

37.3 Special Options

It is possible to obtain output generated by the KANT package using the verbose printing feature of Magma, furthermore, the general style of the output can be changed. It is also possible to alter the precision used for internal real computations.

`SetVerbose(s, n)`

There are a number of verbose flags `s` applying to the functions described in this chapter. The flags and their levels `n` are mentioned in the descriptions of the functions which use them.

Note however, that setting the verbose levels may produce unexpected results since the effective scope of the flags is a little bit vague. Consider the following example:

```
> SetVerbose("MaximalOrder", 1);
> SetVerbose("Factor", 1);
> L := NumberField(PolynomialRing(Rationals()).1^2-10);
Factorize square-free polynomial over Z of degree 2
Deflation factor: 2
Number of deflated factors: 1
Factor inflated polynomial 0 of degree 2
Total factorization time: 0.000
Final irreducibility test factorization:
<x^2 - 10, 0>
> Regulator(L);
order_maximal_sub: called with algo_flag: 0
no algorithm selected
nothing about algebra-splitting selected
nothing about reduced-discriminant selected
nothing about dedekind-test selected
order_maximal_sub: calling order_maximal_sub_sub
order_maximal_sub_sub: called with algo_flag: 16
no algorithm selected
nothing about algebra-splitting selected
```

```
use reduced-discriminant selected
nothing about dedekind-test selected
red disc: f =x^2 - 10
 r_disc = 20
Reduced discriminant: 20
Factorization of reduced discriminant:
2^2 * 5^1
calculation and factorisation of reduced discriminant: 0.01
Factorization of discriminant:
2^3 * 5^1
factors with (possibly) not maximal overorder:
2^3
-----------------------
order_max_p_sub called:
prime: 2, prime_bound: 3, algo_flag: 16
-----------------------
no algorithm selected
nothing about algebra-splitting selected
nothing about dedekind-test selected
---------------------------
order_max_p_sub_sub called:
prime: 2, prime_bound: 3, algo_flag: 89
---------------------------
round2 selected
no algebra-splitting selected
use dedekind-test selected
No split performed ...
(due to user advice or impossible) ...
standard algorithm.
----------------------------
order_max_p_rnd2_sub called:
prime: 2, prime_bound: 3, algo_flag: 89
----------------------------
use dedekind-test selected
Order is already 2-maximal.
1.8184464592320668234836989635607089937862539427689999999
```

The first few lines of output are generated, because the creation of number fields involves a test of irreducibility for the defining polynomial(s).

The next group of lines come from the computation of the maximal order which is used for the regulator computation.

In general the amount of output generated increases with the value supplied. Furthermore, the output corresponding to larger values gets more and more technical.

```
SetKantPrinting(f)
```

Kant-style printing means that integers and rational numbers will be printed as integers and rational numbers. Especially in relative extensions this produces easier to read output - but it is no longer possible to paste the output back into the system again. Turns Kant-style printing on if f is `true` and off if f is `false`.

```
SetKantPrecision(n)
```
```
SetKantPrecision(O, n)
```
```
SetKantPrecision(O, n, m)
```
```
SetKantPrecision(F, n)
```
```
SetKantPrecision(F, n, m)
```

For internal real computations, some number field functions use real arithmetic to a fixed precision; therefore every algebraic field and every order comes equipped with some real-rings.

Initially, the precision (in decimal digits) will be the maximum of the set precision P, 20 and four times the degree of the algebraic field (order) in which calculations are performed. By default $P = 52$.

In addition all unit computations use a second real-ring in which the default precision is always twice the ordinary precision. This ring can be set using the third argument of `SetKantPrecision`.

Furthermore, the computation of the zeroes of the defining polynomial of the field or order uses a third, independent, precision which initially is the maximum of 32 and the (binary) logarithm of the largest coefficient of the defining polynomial.

Note however, that several functions will work automatically with a much larger precision if necessary to guarantee P digits for the user. Generally all functions take care of the necessary precision. Only under rare circumstances will any computation fail because of precision loss.

Certain calculations that fail if the precision is too small may succeed after restarting with increased precision.

If the order O or field F is given, then the precision change will apply only in the context of that order (field).

37.4 Structure Operations

In the lists below F usually refers to an algebraic field, K to a number field and O to an order.

37.4.1 General Functions

Number fields form the MAGMA category `FldNum`, orders form `RngOrd` and their fields of fractions form `FldOrd`. The notional power structures exist as parents of algebraic fields and their orders, no operations are allowed.

`Category(F)` `Parent(F)` `Category(O)` `Parent(O)`

`AssignNames(~K, s)`

Procedure to change the names of the generating elements in the number field K to the contents of the sequence of strings s.

The i-th sequence element will be the name used for the generator of the $(i-1)$-st subfield down from K as determined by the creation of K, the first element being used as the name for the generator of K. In the case where K is defined by more than one polynomial as an absolute extension, the ith sequence element will be the name used for the root of the ith polynomial used in the creation of K.

This procedure only changes the names used in printing the elements of K. It does *not* assign to any identifiers the value of a generator in K; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies K, it is necessary to have a reference `~K` to K in the call to this function.

`Name(K, i)`
`K . i`

Given a number field K, return the element which has the i-th name attached to it, that is, the generator of the $(i-1)$-st subfield down from K as determined by the creation of K. Here i must be in the range $1 \leq i \leq m$, where m is the number of polynomials used in creating K. If K was created using multiple polynomials as an absolute extension, `K.i` will be a root of the ith polynomial used in creating K.

`AssignNames(~F, s)`

Assign the strings in the sequence s to the names of the basis elements of the field of fractions F.

`F . i`
`Name(F, i)`

Return the ith basis element of the field of fractions F.

`O . i`

Return the ith basis element of the order O.

37.4.2 Related Structures

Each order and field has other orders and fields which are related to it in various ways.

`GroundField(F)`

`BaseField(F)`

`CoefficientField(F)`

`CoefficientRing(F)`

Given an algebraic field F, return the algebraic field over which F was defined. For an absolute number field F, the function returns the rational field $\mathbf{Q}$.

`BaseRing(O)`

`CoefficientRing(O)`

Given an order O, this returns the order over which O was defined. For an absolute order O this will be the integers $\mathbf{Z}$.

`AbsoluteField(F)`

Given an algebraic field F, this returns an isomorphic number field L defined as an absolute extension (i.e. over $\mathbf{Q}$).

`AbsoluteOrder(O)`

Given an order O, this returns an isomorphic order O' defined as an order in an absolute extension (over $\mathbf{Q}$).

`SimpleExtension(F)`

`SimpleExtension(O)`

Given an algebraic field F or an order O, this returns an isomorphic field L defined as an absolute simple extension or the Z-isomorphic order in it.

`RelativeField(F, L)`

Given algebraic fields L and F such that Magma knows that F is a subfield of L, return an isomorphic algebraic field M defined as an extension over F.

Example H37E8

It is often desirable to build up a number field by adjoining several algebraic numbers to $\mathbf{Q}$. The following function returns a number field that is the composite field of two given number fields K and L, provided that $K \cap L = \mathbf{Q}$; if K and L have a common subfield larger than $\mathbf{Q}$ the function returns a field with the property that it contains a subfield isomorphic to K as well as a subfield isomorphic to L.

```
> R<x> := PolynomialRing(Integers());
> Composite := function( K, L )
>     T<y> := PolynomialRing( K );
>     f := T!DefiningPolynomial( L );
>     ff := Factorization(f);
>     LKM := NumberField(ff[1][1]);
>     return AbsoluteField(LKM);
> end function;
```

To create, for example, the field $\mathbf{Q}(\sqrt{2}, \sqrt{3}, \sqrt{5})$, the above function should be applied twice:

```
> K := NumberField(x^2-3);
> L := NumberField(x^2-2);
> M := NumberField(x^2-5);
> KL := Composite(K, L);
> S<s> := PolynomialRing(BaseField(KL));
> KLM<w> := Composite(KL, M);
> KLM;
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

Note, that the same field may be constructed with just one call to `NumberField` followed by `AbsoluteField`:

```
> KLM2 := AbsoluteField(NumberField([x^2-3, x^2-2, x^2-5]));
> KLM2;
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

or by

```
> AbsoluteField(ext<Rationals() | [x^2-3, x^2-2, x^2-5]>);
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

In general, however, the resulting polynomials of KLM and $KLM2$ will differ. To see the difference between `SimpleExtension` and `AbsoluteField`, we will create `KLM2` again:

```
> KLM3 := NumberField([x^2-3, x^2-2, x^2-5]: Abs);
> AbsoluteField(KLM3);
Number Field with defining polynomials [ x^2 - 3, x^2 - 2,
    x^2 - 5] over the Rational Field
> SimpleExtension(KLM3);
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

`Simplify(O)`

Given an order O, obtained by a chain of transformations from an equation order E, return an order that is given directly by a single transformation over E.

`LLL(O)`

Given an order O, return an order O' obtained from O by a transformation matrix T, which is returned as a second value. O' will have a LLL-reduced basis.

Example H37E9

Using LLL to reduce the basis of an order is shown.

```
> M := MaximalOrder(x^4-14*x^3+14*x^2-14*x+14);
> L, T := LLL(M);
> L;
Maximal Order, Transformation of M
Transformation Matrix:
[  1   0   0   0]
[ -3   1   0   0]
[  3 -13   1   0]
[ -7   1 -13   1]
> T;
[  1   0   0   0]
[ -3   1   0   0]
[  3 -13   1   0]
[ -7   1 -13   1]
> Basis(M);
[
    M.1,
    M.2,
    M.3,
    M.4
]
> Basis(L, M);
[
    [1, 0, 0, 0],
    [-3, 1, 0, 0],
    [3, -13, 1, 0],
    [-7, 1, -13, 1]
]
> L eq M;
true
```

Even though L and M are considered to be equal because they contain the same elements L has a basis which is LLL reduced but M does not.
Following on from the orders and ideals example (H37E5) we have

```
> R<x> := PolynomialRing(Integers());
> K := NumberField(x^4-420*x^2+40000);
> E := EquationOrder(K);
> O := Round2(E, K);
[ <2, 18>, <5, 8>, <41, 2> ]
index is 2
index is 4
index is 8
index is 4
index is 2
index is 1
index is 5
index is 25
index is 1
index is 1
index of equation order in maximal order is: 64000
> L := LLL(O);
> O;
Transformation of E
Transformation Matrix:
[800   0   0   0]
[  0 400   0   0]
[  0 200  20   0]
[400 180   0   1]
Denominator: 800
> O:Maximal;
   F[1]
    |
   F[2]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     x^4 - 420*x^2 + 40000
Index: 64000/1
Signature: [4, 0]
> L;
Transformation of O
Transformation Matrix:
[-1  0  0  0]
[-1 -1  0  1]
[10  1 -2  0]
[ 5  1 -1  0]
> L:Maximal;
   F[1]
    |
   F[2]
    |
   F[3]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     Given by transformation matrix
F  [ 3]     x^4 - 420*x^2 + 40000
Index: 1/1
Signature: [4, 0]
> Simplify(L):Maximal;
   F[1]
```

```
    |
   F[2]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     x^4 - 420*x^2 + 40000
Index: 64000/1
Signature: [4, 0]
> Simplify(L);
Transformation of E
Transformation Matrix:
[-800    0    0    0]
[-400 -220    0    1]
[8000    0  -40    0]
[4000  200  -20    0]
Denominator: 800
```

`PrimeRing(F)` `PrimeField(F)` `PrimeRing(O)`
`Centre(F)` `Centre(O)`

`Embed(F, L, a)`

Install the embedding of a simple field F in L where the image of the primitive element of F is the element a of L. This embedding will be used in coercing from F into L.

`Embed(F, L, a)`

Install the embedding of the non-simple field F in L where the image of the generating elements of F are in the sequence a of elements of L. This embedding will be used in coercing from F into L.

`EmbeddingMap(F, L)`

Returns the embedding map of F in L if an embedding is known.

Example H37E10

Magma does not recognize two independently created number fields as equal since more than one embedding of a field in a larger field may be possible. To coerce between them then it is convenient to be able to embed them in each other.

```
> k := NumberField(x^2-2);
> l := NumberField(x^2-2);
>  l!k.1;
>> l!k.1;
    ^
Runtime error in '!': Arguments are not compatible
LHS: FldNum
RHS: FldNumElt
> l eq k;
false
> Embed(k, l, l.1);
> l!k.1;
l.1
> Embed(l, k, k.1);
> k!l.1;
k.1
```

`Embed` is useful in specifying the embedding of a field in a larger field.

```
> l<a> := NumberField(x^3-2);
> L<b> := NumberField(x^6+108);
> Root(L!2, 3);
1/18*b^4
> Embed(l, L, $1);
> L!l.1;
1/18*b^4
```

Another embedding would be

```
> Roots(PolynomialRing(L)!DefiningPolynomial(l));
[
    <1/36*(-b^4 - 18*b), 1>,
    <1/36*(-b^4 + 18*b), 1>,
    <1/18*b^4, 1>
]
> Embed(l, L, $1[1][1]);
> L!l.1;
1/36*(-b^4 - 18*b)
```

`Lattice(O)`

`MinkowskiLattice(O)`

Given an absolute order O, returns the lattice determined by the real and complex embeddings of O.

`MinkowskiSpace(F)`

The Minkowski vector space V of the absolute field F as a real vector space, with inner product given by the T_2-norm (`Length`) on F, and by the embedding $F \to V$.

`Completion(K, P)`

`Completion(O, P)`

`comp< K|P >`

`comp< O|P >`

`Precision` RngIntElt *Default :* 50

For an absolute extension K of $\mathbf{Q}$ or O of $\mathbf{Z}$, compute the completion at a prime ideal P which must be either a prime ideal of the maximal order or unramified. The result will be a local field or ring with precision `Precision` or e`*Precision` if the ideal is ramified with ramification degree e.

The returned map is the canonical injection into the completion. It allows pointwise inverse operations.

`Completion(K, P)`

`Precision` RngIntElt *Default :* 50

For an absolute extension K over $\mathbf{Q}$ and a (finite) place P, compute the completion at P. The precision and the map are as described for `Completion`.

`LocalRing(P, prec)`

The completion of `Order(P)` at the prime ideal P up to precision `prec`.

37.4.3 Representing Fields as Vector Spaces

It is possible to express a number field or a field of fractions as a vector space of any subfield using the intrinsics below. Such a construction also allows one to find properties of elements over these subfields.

`Algebra(K, J)`

`Algebra(K, J, S)`

Returns the associative structure constant algebra which is isomorphic to the algebraic field K as an algebra over J. Also returns the isomorphism from K to the algebra mapping w^i to the $i+1$st unit vector of the algebra where w is a primitive element of K.

If a sequence S is given it is taken to be a basis of K over J and the isomorphism will map the ith element of S to the ith unit vector of the algebra.

```
VectorSpace(K, J)
KSpace(K, J)
VectorSpace(K, J, S)
KSpace(K, J, S)
```

The vector space isomorphic to the algebraic field K as a vector space over J and the isomorphism from K to the vector space. The isomorphism maps w^i to the $i + 1$st unit vector of the vector space where w is a primitive element of K.

If S is given, the isomorphism will map the ith element of S to the ith unit vector of the vector space.

Example H37E11

We use the Algebra of a relative number field to obtain the minimal polynomial of an element over a subfield which is not in its coefficient field tower.

```
> K := NumberField([x^2 - 2, x^2 - 3, x^2 - 7]);
> J := AbsoluteField(NumberField([x^2 - 2, x^2 - 7]));
> A, m := Algebra(K, J);
> A;
Associative Algebra of dimension 2 with base ring J
> m;
Mapping from: RngOrd: K to AlgAss: A
> m(K.1);
(1/10*(J.1^3 - 13*J.1)                      0)
> m(K.1^2);
(2 0)
> m(K.2);
(1/470*(83*J.1^3 + 125*J.1^2 - 1419*J.1 - 1735) 1/940*(-24*J.1^3 - 5*J.1^2 +
    382*J.1 + 295))
> m(K.2^2);
(3 0)
> m(K.3);
(1/10*(-J.1^3 + 23*J.1)                      0)
> m(K.3^2);
(7 0)
> A.1 @@ m;
1
> A.2 @@ m;
(($.1 - 1)*$.1 - $.1 - 1)*K.1 + ($.1 + 1)*$.1 + $.1 + 1
>
> r := 5*K.1 - 8*K.2 + K.3;
> m(r);
(1/235*(-238*J.1^3 - 500*J.1^2 + 4689*J.1 + 6940) 1/235*(48*J.1^3 + 10*J.1^2 -
    764*J.1 - 590))
> MinimalPolynomial($1);
$.1^2 + 1/5*(-4*J.1^3 + 42*J.1)*$.1 + 5*J.1^2 - 180
```

```
> Evaluate($1, r);
0
> K:Maximal;
  K
  |
  |
  $1
  |
  |
  $2
  |
  |
  Q
K  : $.1^2 - 2
$1 : $.1^2 - 3
$2 : x^2 - 7
> Parent($3);
Univariate Polynomial Ring over J
> J;
Number Field with defining polynomial $.1^4 - 18*$.1^2 + 25 over the Rational
Field
```

37.4.4 Invariants

Some information describing an order can be retrieved.

`Characteristic(F)` `Characteristic(O)`

`Degree(O)`
`Degree(F)`

Given an algebraic field F, return the degree $[F : G]$ of F over its ground field G. For an order O it returns the relative degree of O over its ground order.

`AbsoluteDegree(O)`
`AbsoluteDegree(F)`

Given an order O or an algebraic field F, return the absolute degree of O over **Z** or F over **Q**.

```
Discriminant(O)
Discriminant(F)
```

Given an extension F of $\mathbf{Q}$, return the discriminant of F. This discriminant is defined to be the discriminant of the order of the field of fractions or the equation order of a number field.

The discriminant of any order O defined over $\mathbf{Z}$ is by definition the discriminant of its basis, where the discriminant of any sequence of elements ω_i from K is defined to be the determinant of the trace matrix of the sequence.

The discriminant of absolute fields and orders is an integer.

The discriminant in a relative extension is the ideal generated by the discriminants of all sequences of elements ω_i from O, where the discriminant of a sequence is defined to be the determinant of its trace matrix. This can only be computed in when the coefficient ring of O is maximal or when O has a power basis.

The discriminant of relative orders is an ideal of the base ring.

```
AbsoluteDiscriminant(O)
```

Given an order O, return the absolute value of the discriminant of O regarded as an order over $\mathbf{Z}$.

```
AbsoluteDiscriminant(K)
```

Given an algebraic field K, return the absolute value of the discriminant of K regarded as an extension of $\mathbf{Q}$.

```
ReducedDiscriminant(O)
ReducedDiscriminant(F)
```

The reduced discriminant of an order O is defined as the maximal elementary divisor (elementary ideal) of the torsion module $O^{\#}/O$ where $O^{\#}$ is the dual module to O with respect to the trace form. The reduced discriminant of an algebraic field is that of the order of a field of fractions and that of the equation order of a number field.

For absolute extensions this is the largest entry of the Smith normal form of the `TraceMatrix`. For relative extensions, in addition to the `TraceMatrix` one has to consider the coefficient ideals.

For orders with a power basis, this is (a generator of) the inverse of the ideal generated by the cofactors X and Y of $Xf + Yf' = 1$ where f is the defining polynomial of the order and f' its first derivative.

```
Regulator(O: parameters)
```

`Current`	BoolElt	*Default :* `false`

```
Regulator(K)
```

Given a number field K or an order O, return the regulator of K or O, as a real number. Note that this will trigger the computation of the maximal order and its unit group if they are not known yet. This only works in an absolute extension.

If `Current` is `true` and a maximal system of independent units is known, then the regulator of that system is returned. In this case no effort is spent to produce a system of fundamental units.

`RegulatorLowerBound(O)`

`RegulatorLowerBound(K)`

Given an order O or number field K, return a lower bound on the regulator of O or K. This only works in an absolute extension.

`Signature(O)`

`Signature(F)`

Given an absolute algebraic field F, or an order O of F, returns two integers, one being the number of real embeddings, the other the number of pairs of complex embeddings of F.

`UnitRank(O)`

`UnitRank(K)`

The unit rank of the order O or the number field K (one less than the number of real embeddings plus number of pairs of complex embeddings).

`Index(O, S)`

The module index of order S in order O, for orders $S \subset O$. O and S must have the same equation order and S must be a suborder of O.

`DefiningPolynomial(F)`

`DefiningPolynomial(O)`

Given an algebraic field F, the polynomial defining F as an extension of its ground field G is returned. For an order O, a integral polynomial is returned that defines O over its coefficient ring.

For non simple extensions, this will return a list of polynomials.

`Zeroes(O, n)`

`Zeros(O, n)`

`Zeroes(F, n)`

`Zeros(F, n)`

Given an absolute algebraic field F or an order O in F, and an integer n, return the zeroes of the defining polynomial of F with a precision of exactly n decimal digits. The function returns a sequence of length the degree of F; all of the real zeroes appear before the complex zeroes.

Example H37E12

The information provided by `Zeros` and `DefiningPolynomial` is illustrated below.

```
> L := NumberField(x^6+108);
> DefiningPolynomial(L);
x^6 + 108
> Zeros(L, 30);
[ 1.88988157484230974715081591089999999994 +
1.09112363597172140356007261419999999977*i,
1.88988157484230974715081591089999999994 -
1.09112363597172140356007261419999999977*i,  0.E-29 +
2.18224727194344280712014522839999999955*i,  0.E-29 -
2.18224727194344280712014522839999999955*i,
-1.88988157484230974715081591089999999994 +
1.09112363597172140356007261419999999977*i,
-1.88988157484230974715081591089999999994 -
1.09112363597172140356007261419999999977*i ]
> l := NumberField(x^3 - 2);
> DefiningPolynomial(l);
x^3 - 2
> Zeros(l, 30);
[ 1.25992104989487316476721060729999999994,
-0.629960524947436582383605303639109999999 +
1.09112363597172140356007261419999999977*i,
-0.629960524947436582383605303639109999999 -
1.09112363597172140356007261419999999977*i ]
```

`Different(O)`

The different of a maximal order $O \subset K$ is defined as the inverse ideal of $\{x \in K \mid \mathrm{Tr}(xO) \subset O\}$.

`Conductor(O)`

The conductor of an order O is the largest ideal of its maximal order that is still contained in O: $\{x \in M \mid xM \subseteq O\}$.

37.4.5 Basis Representation

The basis of an order or algebraic field can be expressed using elements from any compatible ring. Basis related matrices can be formed.

```
Basis(O)
Basis(O, R)
Basis(F)
Basis(F, R)
```

Return the current basis for the order O or algebraic field F over its ground ring as a sequence of elements of its field of fractions or as a sequence of elements of R.

```
IntegralBasis(F)
IntegralBasis(F, R)
```

An integral basis for the algebraic field F is returned as a sequence of elements of F or R if given. This is the same as the basis for the maximal order. Note that the maximal order will be determined (and stored) if necessary.

Example H37E13

The following illustrates how a basis can look different when expressed in a different ring.

```
> f := x^5 + 5*x^4 - 75*x^3 + 250*x^2 + 65625;
> M := MaximalOrder(f);
> M;
Maximal Order of Equation Order with defining polynomial x^5 + 5*x^4 - 75*x^3 +
    250*x^2 + 65625 over its ground order
> Basis(M);
[
    M.1,
    M.2,
    M.3,
    M.4,
    M.5
]
> Basis(NumberField(M));
[
    1,
    $.1,
    $.1^2,
    $.1^3,
    $.1^4
]
> Basis(M, NumberField(M));
[
    1,
    1/5*$.1,
    1/25*$.1^2,
    1/125*$.1^3,
    1/625*$.1^4
]
```

`AbsoluteBasis(K)`

Returns an absolute basis for the algebraic field K, i.e. a basis for K as a $\mathbf{Q}$ vector space. The basis will consist of the products of the basis elements of the intermediate fields. The expansion is done depth-first.

`BasisMatrix(O)`

Given an order O in a number field K of degree n, this returns an $n \times n$ matrix whose i-th row contains the (rational) coefficients for the i-th basis element of O with respect to the power basis of K. Thus, if b_i is the i-th basis element of O,

$$b_i = \sum_{j=1}^{n} M_{ij} \alpha^{j-1}$$

where M is the matrix and α is the generator of K.

The matrix is the same as `TransformationMatrix(O, E)`, where E is the equation order for K, except that the entries of the basis matrix are from the subfield of K, instead of the coefficient ring of the order.

`TransformationMatrix(O, P)`

Returns the transformation matrix for the transformation between the orders O and P with common equation order of degree n. The function returns an $n \times n$ matrix T with integral entries as well as a common integer denominator. The rows of the matrix express the n basis elements of O as a linear combination of the basis elements of P. Hence the effect of multiplying T on the left by a row vector v containing the basis coefficients of an element of O is the row vector $v \cdot T$ expressing the same element of the common number field on the basis for P.

`CoefficientIdeals(O)`

The coefficient ideals of the order O of a relative extension. These are the ideals $\{A_i\}$ of the coefficient ring of O such that for every element e of O, $e = \sum_i a_i * b_i$ where $\{b_i\}$ is the basis returned for O and each $a_i \in A_i$.

Example H37E14

We continue our example of a field of degree 4.
The functions `Basis` and `IntegralBasis` both return a sequence of elements, that can be accessed using the operators for enumerated sequences. Note that if, as in our example, O is the maximal order of K, both functions produce the same output:

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> O := MaximalOrder(K);
> I := IntegralBasis(K);
> B := Basis(O);
> I, B;
[
    1,
    1/2*y,
    1/40*(y^2 + 10*y),
    1/800*(y^3 + 180*y + 400)
]
[
    O.1,
    O.2,
    O.3,
    O.4
]
> Basis(O, K);
[
    1,
    1/2*y,
    1/40*(y^2 + 10*y),
    1/800*(y^3 + 180*y + 400)
]
```

The `BasisMatrix` function makes it possible to move between orders, in the following manner. We may regard orders as free **Z**-modules of rank the degree of the number field. The basis matrix then provides the transformation between the order and the equation order. The function `ElementToSequence` can be used to create module elements.

```
> BM := BasisMatrix(O);
> Mod := RSpace(RationalField(), Degree(K));
> z := O ! y;
> e := z^2-3*z;
> em := Mod ! ElementToSequence(e);
> em;
(  0 -26  40   0)
> f := em*BM;
> f;
```

```
( 0 -3  1  0)
```

So, since f is represented with respect to the basis of the equation order, which is the power basis, we indeed get the original element back. Of course it is much more useful to go in the other direction, using the inverse transformation. We check the result in the last line:

```
> E := EquationOrder(K);
> f := y^3+7;
> fm := Mod ! ElementToSequence(f);
> e := fm*BM^-1;
> e;
(-393 -360    0  800)
> &+[e[i]*B[i] : i in [1 .. Degree(K)] ];
-393/1*O.1 - 360/1*O.2 + 800/1*O.4
> K!$1;
y^3 + 7
```

`MultiplicationTable(O)`

Given an order O of some number field K of degree n, return the multiplication table with respect to the basis of O as a sequence of n matrices of size $n \times n$. The i-th matrix will have as its j-th row the basis representation of $b_i b_j$, where b_i is the i-th basis element for O.

`TraceMatrix(O)`

`TraceMatrix(F)`

Return the trace matrix of an order O or algebraic field F, which has the trace $\mathrm{Tr}(\omega_i \omega_j)$ as its i,j-th entry where the ω_i are the basis for O or F.

Example H37E15

We continue our example of a field of degree 4.
The multiplication table of the order O consists of 4 matrices, such that the i-th 4×4 matrix ($1 \le i \le 4$) determines the multiplication by the i-th basis element of O as a linear transformation with respect to that basis. Thus the third row of $T[2]$ gives the basis coefficients for the product of $B[2]$ and $B[3]$, and we can use the sequence reduction operator to calculate $B[2] * B[3]$ in an alternative way:

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> O := MaximalOrder(K);
> B := Basis(O);
> B[2];
O.2
> T := MultiplicationTable(O);
> T[2];
[  0   1   0   0]
[  0  -5  10   0]
[ -5  -7   5  10]
[-25  -7  15   0]
>  &+[ T[2][3][i]*B[i] : i in [1..4] ];
-5/1*O.1 - 7/1*O.2 + 5/1*O.3 + 10/1*O.4
> B[2]*B[3];
-5/1*O.1 - 7/1*O.2 + 5/1*O.3 + 10/1*O.4
```

The trace matrix may be found either by using the built-in function or by the one-line definition given below (for a field of degree 4):

```
> TraceMatrix(O);
[  4   0  21   2]
[  0 210 105 215]
[ 21 105 173 118]
[  2 215 118 226]
> MatrixRing(RationalField(), 4) ! [Trace(B[i]*B[j]): i, j in [1..4] ];
[  4   0  21   2]
[  0 210 105 215]
[ 21 105 173 118]
[  2 215 118 226]
```

37.4.6 Ring Predicates

Orders and algebraic fields can be tested for having several properties that may hold for general rings.

`N eq O`

Two orders are equal if the transformation matrix taking one to the other is integral and has determinant 1 or -1. For the transformation matrix to exist the orders must have the same number field.

`F eq L`

Returns `true` if and only if the fields F and L, are the same.

No two algebraic fields which have been created independently of each other will be considered equal since it is possible that they can be embedded into a larger field in more than one way.

`IsCommutative(R)` `IsUnitary(R)` `IsFinite(R)`

`IsOrdered(R)` `IsField(R)`

`IsNumberField(R)` `IsAlgebraicField(R)`

`IsEuclideanDomain(F)`

This is not a check for euclidean number fields. This function will always return an error.

`IsSimple(F)`

`IsSimple(O)`

Checks if the field F or the order O is defined as a simple extension over the base ring.

`IsPID(F)` `IsUFD(F)`

`IsPrincipalIdealRing(F)`

Always `true` for fields.

`IsPID(O)` `IsUFD(O)`

`IsPrincipalIdealRing(O)`

Always `false` for orders. Even if the class number is 1, orders are not considered to be PIDs.

`IsDomain(R)`

`F ne L` `O ne N` `O subset P` `K subset L`

`HasComplexConjugate(K)`

This function returns `true` if there is an automorphism in the field K that acts like complex conjugation.

`ComplexConjugate(x)`

For an element x of a field K where `HasComplexConjugate` returns `true` (in particular this includes totally real fields, cyclotomic and quadratic fields and CM-extensions), the conjugate of x is returned.

37.4.7 Order Predicates

Since orders are rings with additional properties, special predicates are applicable.

`IsEquationOrder(O)`

This returns `true` if the basis of the order O is an integral power basis, `false` otherwise.

`IsMaximal(O)`

This returns `true` if the order O in the field F is the maximal order of F, `false` otherwise. The user is warned that this may trigger the computation of the maximal order.

`IsAbsoluteOrder(O)`

Returns `true` iff the order O is a constructed as an absolute extension of **Z**.

`IsWildlyRamified(O)`

Returns `true` iff the order O is wildly ramified, i.e. if there is a prime ideal P of O such that the characteristic of its residue class field divides its ramification index.

`IsTamelyRamified(O)`

Returns `true` iff the order O is not wildly ramified, i.e. if for all prime ideals P of O the characteristic of its residue class field does not divide the ramification index.

`IsUnramified(O)`

Returns `true` iff the order O is unramified at the *finite* places.

37.4.8 Field Predicates

Here all the predicates that are specific to algebraic fields are listed.

`IsIsomorphic(F, L)`

Given two algebraic fields F and L, this returns `true` as well as an isomorphism $F \to L$, if F and L are isomorphic, and it returns `false` otherwise.

`IsSubfield(F, L)`

Given two algebraic fields F and L, this returns `true` as well as an embedding $F \hookrightarrow L$, if F is a subfield of L, and it returns `false` otherwise.

`IsNormal(F)`

Returns `true` if and only if the algebraic field F is a normal extension. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois group computation.

`IsAbelian(F)`

Returns `true` if and only if the algebraic field F is a normal extension with abelian Galois group. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois Group computation.

`IsCyclic(F)`

Returns `true` if and only if the algebraic field F is a normal extension with cyclic Galois group. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois and automorphism group.

`IsAbsoluteField(K)`

Returns `true` iff the algebraic field K is a constructed as an absolute extension of $\mathbf{Q}$.

```
IsWildlyRamified(K)
```

Returns `true` iff the algebraic field K is wildly ramified, i.e. if there is a prime ideal P of K (its maximal order) such that the characteristic of its residue class field divides the ramification index.

```
IsTamelyRamified(K)
```

Returns `true` iff the algebraic field K is not wildly ramified, i.e. if for all prime ideals P of the maximal order of K, the characteristic of its residue class field does not divide the ramification index.

```
IsUnramified(K)
```

Returns `true` iff the algebraic field K is unramified at the *finite* places.

```
IsQuadratic(K)
```

If the number field K is quadratic, return `true` and an isomorphic quadratic field.

```
IsTotallyReal(K)
```

Tests if the number field F is totally real, ie. if all infinite places are real. For absolute fields this is equivalent to the defining polynomial having only real roots.

37.4.9 Setting Properties of Orders

Several properties of orders can be set to be known and/or to have a given value.

```
SetOrderMaximal(O, b)
```

Set the order O to be maximal if b is `true` or known to be non maximal if b is `false`.

```
SetOrderTorsionUnit(O, e, r)
```

Set the torsion unit of the order O to be the element e with order r.

```
SetOrderUnitsAreFundamental(O)
```

Mark the currently known units of the order O to be fundamental.

37.5 Element Operations

37.5.1 Parent and Category

`Parent(a)` `Parent(w)` `Category(a)` `Category(w)`

37.5.2 Arithmetic

The table below lists the generic arithmetic functions on algebraic field and order elements. Note that automatic coercion ensures that the binary operations `+`, `-`, `*`, and `/` may be applied to an element of an algebraic field and an element of one of its orders; the result will be an algebraic field element. Since division of order elements does not generally result in an order element, the operation `/` applied to two elements of an order returns an element in the field of fractions of the order; similarly if the exponent k in `a^k` is negative.

For finding the value of an element mod an ideal or the inverse of an element mod an ideal see Section 37.9.6.

`+ a` `- a`

`a + b` `a - b` `a * b` `a / b` `a ^ k`

`w div v`

The quotient of the order element w by the order element v; v must divide w exactly, (v and w must be elements of the same order.)

`Modexp(a, n, m)`

Given a non-negative integer n and an integer m greater than 1, this function returns the modular power $a^n \ mod\ m$ of the order element a.

`Sqrt(a)`
`SquareRoot(a)`

Returns the square root of the element a if it exists in the order or field containing a.

`Root(a, n)`

Returns the n-th root of the element a if it exists in the order or field containing a.

`IsPower(a, k)`
`IsSquare(a)`

Return `true` if the element a is a kth power, (respectively square) and the root in the order or field containing a if so.

`Denominator(a)`

Returns the denominator of the element a, that is the least common multiple of the denominators of the coefficients of a.

`Numerator(a)`

Returns the numerator of the element a, that is the element multiplied by its denominator.

`Qround(E, M)`

`ContFrac` BoolElt *Default :* `true`

Finds an approximation of the field element E where the denominator is bounded by the integer M. If `ContFrac` is `true`, the approximation is computed by applying the continued fraction algorithm to the coefficients of E viewed over Q.

37.5.3 Equality and Membership

Elements may also be tested for whether they lie in an ideal of an order. See Section 37.9.5.

`a eq b` `a ne b`

`a in F`

37.5.4 Predicates on Elements

In addition to the generic predicates `IsMinusOne`, `IsZero` and `IsOne`, the predicates `IsIntegral` and `IsPrimitive` are defined on elements of algebraic fields and orders.

`IsIntegral(a)`

Returns `true` if the element a of an algebraic field F or of an order in F is contained in the ring of integers of F, `false` otherwise. This is vacuously true for order elements. We use the minimal polynomial to determine the answer, which means that the calculation of the maximal order is *not* triggered if it is not known yet. When a is a field element a denominator d such that $d*a$ is integral is also returned on request.

`IsPrimitive(a)`

Returns `true` if the element a of the algebraic field F or one of its orders O generates F.

`IsTorsionUnit(w)`

Returns `true` if and only if the order element w is a unit of finite order.

`IsPower(w, n)`

Given an element w in an order O and an integer $n > 1$, this function returns `true` if and only if there exists an element $v \in O$ such that $w = v^n$; if `true`, such an element v is returned as well.

`IsTotallyPositive(a)`

`IsTotallyPositive(a)`

Returnes `true` iff all real embeddings of the element a are positive. For elements in absolute fields this is equivalent to all real conjugates being positive.

`IsZero(a)` `IsOne(a)`

`IsMinusOne(a)` `IsUnit(a)`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsZeroDivisor(a)` `IsRegular(a)`

`IsIrreducible(a)` `IsPrime(a)`

37.5.5 Finding Special Elements

Generators of fields can be retrieved.

`K . 1`

Return the image α of x in $G[x]/f$ where f is the first defining polynomial of K and G is the base field of K.

In case of simple extensions this will be a primitive element.

`PrimitiveElement(K)`

`PrimitiveElement(F)`

Returns a primitive element for the simple algebraic field, that is an element whose minimal polynomial has the same degree as the field. For a number field K this is $K.1$ but for a field of fractions this is $F!K.1$ where K is the number field of F.

For non-simple fields, a random element is returned.

`Generators(K)`

The list of generators of K over its coefficient field, that is a sequence containing a root of each defining polynomial is returned.

`Generators(K, k)`

A list of generators of K over k is returned. That is a sequence containing a root of each defining polynomial for K and its subfield down to the level of k is returned.

`PrimitiveElement(O)`

Returns a primitive element for the field of fractions of the order O, that is an element whose minimal polynomial has the same degree as the field.

37.5.6 Real and Complex Valued Functions

The functions here return (sequences of) real or complex numbers. The precision of these numbers is governed by the appropriate order's or field's internal precision. See Section 37.3 for more information.

```
AbsoluteValues(a)
```

Return a sequence of length $r_1 + r_2$ of the real absolute values of the conjugates of the element a. The first r_1 values are the absolute values of the real embeddings of the element, the next r_2 are the lengths of the complex embeddings with their weight factors. That is, if the real conjugates of a are w_i, for $1 \le i \le r_1$, and the complex conjugates of a are $x_i \pm iy_i$ (for $1 \le i \le r_2$), then `AbsoluteValues` returns $[|w_1|, \ldots, |w_{r_1}|, \sqrt{\frac{x_{r_1+1}^2+y_{r_1+1}^2}{2}}, \ldots \sqrt{\frac{x_{r_1+r_2}^2+y_{r_1+r_2}^2}{2}}]$.

```
AbsoluteLogarithmicHeight(a)
```

Let P be the minimal polynomial of the element a over $\mathbf{Z}$, with leading coefficient a_0 and roots $\alpha_1, \ldots, \alpha_n$. Then the absolute logarithmic height is defined to be

$$h(\alpha) = \frac{1}{n} \log(a_0 \prod_{j=1}^{n} \max(1, |\alpha_j|)).$$

```
Conjugates(a)
```

The real and complex conjugates of the given algebraic number a, as a sequence of n complex numbers. The r_1 real conjugates appear first, and are followed by r_2 pairs of complex conjugates. The field should be an absolute extension. The ordering of the conjugates is consistent for elements of the same field (or even for elements of different fields that have the same defining polynomial).

```
Conjugate(a, k)
```

Equivalent to `Conjugates(a)[k]`.

```
Conjugate(a, l)
```

Let $l := [l_1, \ldots, l_n]$ be a sequence of positive integers and assume that the field K, the parent of a is given as a tower with n steps, $\mathbf{Q} \subseteq K_1 \subseteq \ldots \subseteq K_n = K$. This function computes the image of a in $\mathbf{C}$ or $\mathbf{R}$ under the embedding determined by l, that is under embedding obtained by extending the l_1 embedding of K_1 to K_2, then extending the l_2nd of those embeddings to K_3, ...

```
Length(a)
```

Return the T_2-norm of the element a, which is a real number. This equals the sum of the (complex) norms of the conjugates of a.

`Logs(a)`

Return the sequence of length $r_1 + r_2$ of logarithms of the absolute values of the conjugates of a number field or order element $a \neq 0$.

`CoefficientHeight(E)`

`CoefficientHeight(E)`

Computes the coefficient height of the element E, that is for an element of an absolute field it returns the maximum of the denominator and the largest coefficient wrt. to the basis of the parent. For elements in relative extensions, it returns the maximal coefficient height of all the coefficients wrt. the basis of the parent.

This function indicates in some way the difficulty of operations involving this element.

`CoefficientLength(E)`

`CoefficientLength(E)`

Computes the coefficient length of the element E, that is for an element of an absolute field it returns the sum of the denominator and the absolute values of all coefficients wrt. to the basis of the parent. For elements in relative extensions, it returns the sum of the coefficient length of all the coefficients wrt. the basis of the parent.

This function gives an indication on the amount of memory occupied by this element.

Example H37E16

Using the functions `Conjugates` and `Basis`, it is easy to write an alternative discriminant function.

```
> disc := func< O | Determinant( MatrixAlgebra(ComplexField(20), Degree(O) )
>       ! [ Conjugates(Basis(O)[i])[j] : i, j in [1 .. Degree(O)] ] )^2 >;
> R<x> := PolynomialRing(Integers());
> O := MaximalOrder(NumberField(x^4 - 420*x^2 + 40000));
> disc(O);
42025
> Discriminant(O);
42025
```

Thus, the new discriminant function returns a complex approximation to the built-in function `Discriminant`, giving the above result for the maximal order O of the previous example.
Here is an alternative way of getting the T_2 norm returned by `Length`, using the complex `Norm` function, together with the `Conjugates` function.

```
> norm := func< a | &+[ Norm(Conjugates(a)[i]) : \
>                       i in [1 .. Degree(Parent(a))] ] >;
```

37.5.7 Norm, Trace, and Minimal Polynomial

The norm, trace and minimal polynomial of order and algebraic field elements can be calculated both with respect to the coefficient ring and to **Z** or **Q**.

`Norm(a)`

`Norm(a, R)`

The relative norm $\mathrm{N_{L/F}(a)}$ over F of the element a of L where F is the field or order over which L is defined as an extension. If R is given the norm is calculated over R. In this case, R must occur as a coefficient ring somewhere in the tower under L.

`AbsoluteNorm(a)`

`NormAbs(a)`

The absolute norm $\mathrm{N_{L/\mathbf{Q}}(a)}$ over **Q** of the element a of L (or one of its orders).

`Trace(a)`

`Trace(a, R)`

The relative trace $\mathrm{Tr}_{L/F}(a)$ over F of the element a of L where F is the field or order over which L is defined as an extension. If R is given the trace is computed over R. In this case, R must occur as a coefficient ring somewhere in the tower under L.

`AbsoluteTrace(a)`

`TraceAbs(a)`

The absolute trace $\mathrm{Tr}_{L/\mathbf{Q}}(a)$ over **Q** of the element a of L (or one of its orders).

`CharacteristicPolynomial(a)`

`CharacteristicPolynomial(a, R)`

Given an element a from an algebraic field or order L, returns the characteristic polynomial of the element over R if given or the subfield or suborder F otherwise where F is the field or order over which L is defined as an extension.

`AbsoluteCharacteristicPolynomial(a)`

Given an element a from an algebraic field or one of its orders, this function returns the characteristic polynomial of the element. For field elements the polynomial will have coefficients in the rational field, for order elements the coefficients will be in the ring of integers.

`MinimalPolynomial(a)`

`MinimalPolynomial(a, R)`

Given an element a from an algebraic field or order L, returns the minimal polynomial of the element over R if given otherwise the subfield or suborder F where F is the field or order over which L is defined as an extension.

`AbsoluteMinimalPolynomial(a)`

Given an element a from an algebraic field or one of its orders, this function returns the minimal polynomial of the element. For field elements the polynomial will have coefficients in the rational field, for order elements the coefficients will be in the ring of integers.

`RepresentationMatrix(a)`

`RepresentationMatrix(a, R)`

Return the representation matrix of a, that is, the matrix which represents the linear map given by multiplication by a. If a is an order element, this matrix is with respect to the basis for the order; if a is an algebraic field element, the basis for the field is used. The ith row of the representation matrix gives the coefficients of aw_i with respect to the basis $w_1, \ldots, w_n$.

If R is given the matrix is over R and with respect to the basis of the order or field over R.

`AbsoluteRepresentationMatrix(a)`

Return the representation matrix of a relative to the $\mathbf{Q}$-basis of the field constructed using products of the basis elements, where a is an element of the relative number field L.

Let $L_i := \sum L_{i-1}\omega_{i,j}$, $L := L_n$ and $L_0 := \mathbf{Q}$. Then the representation matrix is computed with respect to the $\mathbf{Q}$-basis $(\prod_j \omega_{i_j,j})_{i\in I}$ consisting of products of basis elements of the different levels.

Example H37E17

We create the norm, trace, minimal polynomial and representation matrix of the element $\alpha/2$ in the quartic field $\mathbf{Q}(\alpha)$.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^4-420*x^2+40000);
> z := y/2;
> Norm(z), Trace(z);
2500 0
> MinimalPolynomial(z);
ext<Q|>.1^4 - 105*ext<Q|>.1^2 + 2500
> RepresentationMatrix(z);
[     0   1/2     0     0]
[     0     0   1/2     0]
[     0     0     0   1/2]
[-20000     0   210     0]
```

The awkwardness of the printing of the minimal polynomial above can be overcome by providing a parent for the polynomial, keeping in mind that it is a univariate polynomial over the rationals:

```
> P<t> := PolynomialRing(RationalField());
> MinimalPolynomial(z);
```

```
t^4 - 105*t^2 + 2500
```

37.5.8 Other Functions

Elements can be represented by sequences and matrices. Valuation can also be calculated.

`ElementToSequence(a)`

`Eltseq(a)`

For an element a of an algebraic field F, a sequence of coefficients of length degree of F with respect to the basis is returned. For an element of an order O, the sequence of coefficients of the element with respect to the basis of O are returned.

Note however that the universe of the sequence if always a field since in general in relative extensions integral coefficients cannot be achieved.

`Eltseq(E, k)`

For an algebraic number $E \in K$ and a ring k which occurs somewhere in the defining tower for K, return the list of coefficients of E over k, that is, apply `Eltseq` to E and to its coefficients until the list is over k.

`Flat(e)`

The coefficients of the algebraic field element e wrt. to the canonical Q basis for its field. This is performed by iterating `Eltseq` until the coefficients are rational numbers. For number field elements the coefficients obtained match the coefficients wrt. to `AbsoluteBasis`.

`a[i]`

The coefficient of the ith basis element in the algebraic field or order element a.

`ProductRepresentation(a)`

Return sequences P and E such that the product of elements in P to the corresponding exponents in E is the algebraic number a.

`ProductRepresentation(P, E)`

`PowerProduct(P, E)`

Return the element a of the universe of the sequence P such that a is the product of elements of P to the corresponding exponents in the sequence E.

`Valuation(w, I)`

Given a prime ideal I and an element w of an order or algebraic field, this function returns the valuation $v_I(w)$ of w with respect to I; this valuation will be a non-negative integer. Ideals are discussed in Section 37.9.

`Decomposition(a)`

`Decomposition(a)`

The factorization of the order or algebraic field element a into prime ideals.

`Divisors(a)`

For an element a in a maximal order return a sequence containing (up to units) all the elements which divide a. The elements of the sequence will be generators for all principal ideals returned by `Divisors(Parent(a)*a)`.

`Index(a)`

The index of the module $\mathbf{Z}[a]$ in O where a lies in O, an order over $\mathbf{Z}$. If a is not a primitive element the index is infinite.

`Different(a)`

The different of the element a of an order of a number field.

37.6 Ideal Class Groups

This section describes the functions related to finding class groups and class numbers for (the maximal order O of) an absolute number field.

The method usually employed is the *relation* method ([Heß96, Coh93]), basically consisting of the following steps. In the first step a list of prime ideals of norm below a given bound is generated, the *factor basis*. In the second step a search is conducted to find in each of the prime ideals a few elements for which the principal ideals they generate factor completely over the factor basis. Using these *relations*, a generating set for the ideal class group is derived (via matrix echelonization), and in the final step it is verified that the correct orders for the generators have been found.

To determine the class group or class number correctly one has to make sure that all ideals having norm smaller than the Minkowski bound or smaller than the Bach bound, if one assumes the generalized Riemann hypothesis, are taken into consideration, and that the final stage, which may be time consuming, is properly executed. Optional arguments allow the user to override these, but correctness of such results can not be guaranteed.

It should be stressed that by default a guaranteed result is computed using the Minkowski bound. Thus even for innocent looking fields it may take considerable time. In comparison, pari (as of version 2.0) will by default use a much smaller bound giving results that are not guaranteed even under GRH. On the other hand, it will be much faster. It is possible to use the same bounds in Magma. In this case the running times will be similar.

When the discriminant of an order is very large (above 10^{30}) a sieving method developed by [Bia] is used. Using this method the relations are derived using an analogue of the number field sieve. Lattice sieving and the special-Q are also used. The verbose flag `"ClassGroupSieve"` can be set to view information about the computation.

Once a class group computation has been completed, the results are stored with the order.

All functions mentioned in this section support the verbose flag `ClassGroup` up to a maximum value of 5.

Ideals in MAGMA are discussed in Section 37.9.

`DegreeOnePrimeIdeals(O, B)`

Given an order O as well as a positive integer bound B, return a sequence consisting of all prime ideals in O whose norm is a rational prime not exceeding the bound B.

`ClassGroup(O: parameters)`

`ClassGroup(K: parameters)`

`Bound`	RNGINTELT	*Default :* MinkowskiBound
`Proof`	MONSTGELT	*Default :* "*Full*"
`Enum`	BOOLELT	*Default :* `true`
`Al`	MONSTGELT	*Default :* "*Automatic*"
Verbose	`ClassGroup`	*Maximum :* 5
Verbose	`ClassGroupSieve`	*Maximum :* 5

The group of ideal classes for the ring of integers O of the number field K is returned as an abelian group, together with a map from this abstract group to O. The map admits inverses and can therefore be used to compute "discrete logarithms" for the class group.

With the default values for the optional parameters the Minkowski bound is used and the last step of the algorithm verifies correctness (see the explanation above), hence a fully proven result is returned.

If `Bound` is set to some positive integer M, M is used instead of the Minkowski bound. The validity of the result still depends on the `"Proof"` parameter.

If `Proof := "GRH"`, everything remains as in the default case except that a bound based on the GRH is used to replace the Minkowski bound. This bound may be enlarged setting the `Bound` parameter accordingly. The result will hence be correct under the GRH.

If `Proof := "Bound"`, the computation stops if an independent set of relations between the prime ideals below the chosen bound is found. The relations may not be maximal.

If `Proof := "Subgroup"`, a maximal subset of the relations is constructed. In terms of the result, this means that the group returned will be a subgroup of the class group (i.e. the list of prime ideals considered may be to small).

If `Proof := "Full"` (the default) a guaranteed result is computed. This is equivalent to `Bound := MinkowskiBound(K)` and `Proof := "Subgroup"`.

If only `Bound` is given, the `Proof` defaults to `"Subgroup"`.

Finally, giving `Proof := "Current"` is the same as repeating the last call to `ClassGroup()`, but without the need to explicitly restate the value of `Proof` or `Bound`. If there was no prior call to `ClassGroup`, a fully proven computation will be carried out.

If `Enum := false`, then instead of enumerating short elements to get relations, Magma will use random linear combinations of a reduced basis instead. For "small" fields this will typically slow down the computations, but for large fields it is sometimes not possible to find any point using enumeration so that this is necessary for "large" fields. Unfortunately, there is no known criterion to decide beforehand if a field is "large" or "small".

If `Al` is set to `"Sieve"` (regardless of the size of the discriminant) or the discriminant of O is greater than 10^{30} then the sieving method developed by [Bia] is used. If `Al` is set to `"NoSieve"` then this sieving method will not be used regardless of the size of the discriminant.

`RingClassGroup(O)`

`PicardGroup(O)`

For a (possibly non-maximal) order O, compute the ring class group (Picard group) of O, ie. the group of invertible ideals in O modulo pricipal ideals. The algorithm and its implementation are due to Klüners and Pauli, [PK05].

`ConditionalClassGroup(O)`

`ConditionalClassGroup(K)`

The class group of the order O or the number field K assuming the generalized Riemann hypothesis.

`ClassGroupPrimeRepresentatives(O, I)`

For the maximal order O of some absolute number field k and an ideal I of O, compute a set of prime ideals in O that are coprime to I and represent all ideal classes. The map, mapping elements of the class group to the primes representing the ideal class is returned.

`ClassNumber(O:` *parameters*`)`

`ClassNumber(K:` *parameters*`)`

`Bound`	RngIntElt	*Default :* MinkowskiBound
`Proof`	MonStgElt	*Default :* "*Full*"
`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`ClassGroup`	*Maximum :* 5
Verbose	`ClassGroupSieve`	*Maximum :* 5

Return the class number of the ring of integers O of a number field K. The options for the parameters are the same as for `ClassGroup`.

```
BachBound(K)
BachBound(O)
```

An integral upper bound for norms of generators of the ideal class group for the number field K or the maximal order O assuming the generalized Riemann hypothesis.

```
MinkowskiBound(K)
MinkowskiBound(O)
```

An unconditional integral upper bound for norms of the generators of the ideal class group for the number field K or the maximal order O.

```
FactorBasis(K, B)
FactorBasis(O, B)
```

Given the maximal order O, or a number field K with maximal order O, this function returns a sequence of prime ideals of norm less than a given bound B.

```
FactorBasis(O)
```

Given the maximal order O where the class group has previously been computed, this function returns a sequence of prime ideals that have been used as factor basis for the class group computation. In addition the used upper bound for the factor basis is returned. This bound can be different from the bound passed in using the `Bound := bound` parameter.

```
RelationMatrix(K, B)
RelationMatrix(O, B)
```

Given a maximal order O, or a number field K with maximal order O, generate relations for each prime ideal in the factor basis for O with bound B on the norms of the ideals. The relations are given by rows in a matrix. If at some stage the relations generate the trivial group, no more relations are generated.

```
RelationMatrix(O)
```

Given a maximal order O where the class group has been computed previously, the resulting relation matrix is returned.

```
Relations(O)
```

Given a maximal order O where the class group has been computed previously, the vector containing the order elements used to compute the class group is returned.

```
ClassGroupCyclicFactorGenerators(O)
```

Let a_i be the generators for the cyclic factors of the class group of O. This function returns generators for $a_i^{c_i}$ where c_i is the order of a_i in the class group.

Example H37E18

We give an example of a class group calculation, illustrating some of the functions.

```
> R<x> := PolynomialRing(Integers());
> O := MaximalOrder(x^2-10);
> C, m := ClassGroup(O);
> C;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*C.1 = 0
> m(C.1);
Prime Ideal of O
Two element generators:
    [2, 0]
    [0, 1]
> p := Decomposition(O, 31)[1][1];
> p;
Prime Ideal of O
Two element generators:
    [31, 0]
    [45, 1]
> p @@ m;
0
> IsPrincipal(p);
true
> p := Decomposition(O, 37)[1][1];
> p @@ m;
C.1
> IsPrincipal(p);
false
> MinkowskiBound(O);
3
> F, B := FactorBasis(O);
> B;
33
```

Even though the MinkowskiBound is only 3, we take 33 as the bound. The reason for this behaviour is that the factor base has to have at least some elements (about 20) if it is non-empty. Otherwise the search for relations is hopeless.

```
> r := Relations(O);
> M := RelationMatrix(O);
> [ Valuation(r[1][1], x) : x in F];
[ 0, 0, 0, 0, 0, 1, 1, 0 ]
> M[1];
```

```
(0 0 0 0 0 1 1 0)
```

As one can see, the `RelationMatrix` basically stores the valuation of the elements of `Relations` at the prime ideal contained in `FactorBasis`.

```
> f,g := IsPrincipal(m(C.1)^2);
> f;
true
> g;
[2, 0]
> ClassGroupCyclicFactorGenerators(O);
[
    [2, 0]
]
```

Now we will consider some larger fields to demonstrate the effect of the `"Bound"` parameter:

```
> K := NumberField(x^5-14*x^4+14*x^3-14*x^2+14*x-14);
> MinkowskiBound(K);
21106
> BachBound(K);
7783
> time ClassGroup(K);
Abelian Group isomorphic to Z/10
Defined on 1 generator
Relations:
    10*$.1 = 0
Mapping from: Abelian Group isomorphic to Z/10
Defined on 1 generator
Relations:
    10*$.1 = 0 to Set of ideals of Maximal Equation Order with defining
polynomial x^5 - 14*x^4 + 14*x^3 - 14*x^2 + 14*x - 14 over Z
Time: 14.600
```

Note, that is an unconditional result. As one can see, the `BachBound` (proven under GRH) is much smaller than `MinkowskiBound`. The difference shows up in the running time:

```
> K := NumberField(x^5-14*x^4+14*x^3-14*x^2+14*x-14);
> time _, _ := ClassGroup(K: Bound := BachBound(K));
Time: 7.080
```

In comparison, pari (2.0.20) uses an even smaller bound as default, namely `BachBound(K)/40` for fields of degree > 2 and `BachBound(K)/20` for quadratics.

```
> K := NumberField(x^5-14*x^4+14*x^3-14*x^2+14*x-14);
> time _, _ := ClassGroup(K: Bound := Floor(BachBound(K)/40));
Time: 1.300
```

In comparison, `bnfinit` in pari (version 2.0.20) takes about 1.4 seconds for this example.
Note, that in general one cannot use arbitrarily small bounds. If they are too small, the computation time will increase again as the system will not be able to find any relations.

Example H37E19

We give some examples of class group computations using the sieving algorithm.

```
> R<x> := PolynomialRing(IntegerRing());
> f := x^3 + 3592347*x^2 - 6*x + 3989864;
> K<y> := NumberField(f);
> time G, m := ClassGroup(K : Al := "Sieve", Proof := "GRH");
Time: 3.200
> G, m;
Abelian Group isomorphic to Z/6
Defined on 1 generator
Relations:
    6*G.1 = 0
Mapping from: GrpAb: G to Set of ideals of Maximal Order of Equation Order with
defining polynomial x^3 + 3592347*x^2 - 6*x + 3989864 over Z
> m(G.1);
Ideal
Two element generators:
    [4373, 0, 0]
    [2118, 1, 0]
> $1 @@ m;
G.1
> IsPrincipal($2^6);
true
> f:= x^5 + 3330*x^3 - 50*x + 107392;
> K<y> := NumberField(f);
> time G, m := ClassGroup(K : Al := "Sieve", Proof := "GRH");
Time: 9.930
> G, m;
Abelian Group of order 1
Mapping from: GrpAb: G to Set of ideals of Maximal Order of Equation Order with
defining polynomial x^5 + 3330*x^3 - 50*x + 107392 over Z
```

It is also possible to drive the class group computation “by hand”, that is one can call the individual parts one by one to for example re-create a class group computation:

`FactorBasisCreate(O,B)`

Creates a class group process by computing a factor basis containing all ideals of norm $\leq B$ in the order O and returning this factor basis.

`EulerProduct(O, B)`

Computes an approximation to the Euler product for the order O using only prime ideals over prime numbers of norm $\leq B$.

`AddRelation(E)`

Adds a relation (order element) E to the class group process of the parent of E. This function returns `true` exactly when the element factors over the factor basis and if the new relation matrix is of full rank.

`EvaluateClassGroup(O)`

Finalizes a class group process for the order O, that is, it computes (if possible) the class group structure based on the current relation matrix and a basis of the unit group generated by the nullspace of the relation matrix.

This function returns `true` when the class group is determined by the current data. In order to use this function, one has to create a factor basis and then add enough relations.

`CompleteClassGroup(O)`

This function completes an already started class group process for the order O by using the internal functions to look for relations until the class group can be determined.

`FactorBasisVerify(O, L, U)`

This function verifies the "completeness" of the current factor basis for the order O with respect to the prime ideals of norm between L and U. That is, for all prime ideals which norm is between L and U, the function tries to find a relation between the new prime ideal and the prime ideals already in the factor basis. If successful, this means that the new prime ideal does not contribute anything to the class group that is not already known and can therefore safely be ignored.

This function does not return if unsuccessful.

`ClassGroupSetUseMemory(O, f)`

For an order O where the class group is already computed, decide if results of the discrete logarithm computation for the class group are stored. For example if the order O is going to be used extensively as a coefficient ring for class field computations, then every time discrete logatrithms of ray class groups are computed, a discrete logarithm computation in the class group is triggered. In particular when investigating the cohomology of various extensions over O, this involves testing the same ideals over and over again. Setting the flag f to `true` will help to keep computation times down - at the expense of additional use of memory. This functionality is disabled by default.

`ClassGroupGetUseMemory(O)`

For an order O where the class group has been computed check if discrete logarithm values should be stored or not.

37.6.1 Setting the Class Group Bounds Globally

It is possible to preset the bounds to be used in all class group computations. Two bounds can be specified: the first controls the size of the factor base used in the first stage of the class group computation, and the second is the bound used in the checking stage (this controls the level of rigour of the computation).

The bounds may be specified using either of the intrinsics below; `SetClassGroupBounds` is a simple special case of `SetClassGroupBoundMaps`, provided for the user's convenience.

`SetClassGroupBounds(n)`

`SetClassGroupBounds(string)`

This determines the bounds that will be used in all subsequent calls to `ClassGroup`. The argument can be an integer n (then the bounds are both set to this constant). Alternatively the argument can be a string: either "GRH" (then the bounds will guarantee correctness assuming GRH) or "PARI" (this is intended to give roughly the same level of rigour as PARI).

`SetClassGroupBoundMaps(f1, f2)`

This determines the bounds that will be used in all subsequent calls to `ClassGroup`. The arguments should be maps from `PowerStructure(RngOrd)` (which is the parent object of all orders in number fields) to integers.

The bounds used when `ClassGroup` is called for a number field will be the bounds for the maximal order of that field.

Example H37E20

We select some bounds which will then be used in all calls to `ClassGroup`. (The class group computations will be rigorous, but will use a relatively small factor base for the first part of the computation).

```
> map1 := map< PowerStructure(RngOrd) -> Integers() |
>                             order :-> BachBound(order) div 10 >;
> map2 := map< PowerStructure(RngOrd) -> Integers() |
>                             order :-> MinkowskiBound(order) >;
> SetClassGroupBoundMaps( map1, map2);
```

37.7 Unit Groups

The relation method, outlined in the previous section, can also be used for unit group calculations. Therefore, unit group calculations including those triggered as a side effect may cause the creation of factor bases and relations. Other methods such as Dirichlet's method are also implemented which may be faster in certain circumstances. Descriptions of the algorithms can be found in [PZ89], (pp. 343–344), and in [Poh93]. These methods will only work for absolute extensions.

In general, the unit group related functions support a verbose flag `UnitGroup` up to a maximum of 6.

`UnitGroup(O)`

`MultiplicativeGroup(O)`

`UnitGroup(K)`

`MultiplicativeGroup(K)`

`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`UnitGroup`	*Maximum :* 6

Given an order O in a number field, this function returns an (abstract) abelian group U, as well as a bijection m between U and the units of the order. The unit group consists of the torsion subgroup, generated by the image `m(U.1)` and a free part, generated in O by the images `m(U.i)` for $2 \le i \le r_1 + r_2$.

If the argument to this function is a number field K, the unit group of its maximal order is returned. Note that the maximal order may have to be determined first.

The parameter `Al` can be used to specify an algorithm. It should be one of `"Automatic"`, (default, a choice will be made for the user) `"ClassGroup"`, `"Dirichlet"`, `"Mixed"` (the best known Dirichlet method), `"Relation"` or `"Short"` (which is a variation of `"Mixed"`). In the case of real quadratic fields, a continued fraction algorithm is available, `"ContFrac"`.

`UnitGroupAsSubgroup(O)`

For a (possibly non-maximal) order O in some absolute field K, return the unit group of O as a subgroup of the unit group of the maximal order of O. The algorithm and its implementatio is due to Klüners and Pauli, [PK05].

`TorsionUnitGroup(O)`

`TorsionUnitGroup(K)`

The torsion subgroup of the unit group of the order O, or, in case of a number field K, of its maximal order O. The torsion subgroup is returned as an abelian group T, together with a map m from the group to the order O. The torsion subgroup will be cyclic, and is generated by `m(T.1)`.

```
IndependentUnits(O)
IndependentUnits(K)
```

`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`UnitGroup`	*Maximum :* 6

Given an order O, this function returns a sequence of independent units; they generate a subgroup of finite index in the full unit group. Given a number field K, the function is applied to the maximal order of K. The function returns an abelian group generated by the independent units as well as a homomorphism from the group to the order.

The parameter `Al` can be used to specify an algorithm. It should be one of `"Automatic"`, `"ClassGroup"`, `"Dirichlet"`, `"Mixed"` (the best known Dirichlet-like method), `"Relation"`. or `"Short"` (which is a variation of `"Mixed"`). In the case of real quadratic fields, a continued fraction algorithm is available, `"ContFrac"`.

```
pFundamentalUnits(O, p)
pFundamentalUnits(K, p)
```

`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`UnitGroup`	*Maximum :* 6

Given an order O in a number field, this function returns an (abstract) abelian group U, as well as a map m from U to the order. U will be a subgroup (of finite index) of the unit group G such that p does not divide the index $(G : U)$ where p is the prime number given. If a field K is given rather than an order, the above is computed for the maximal order of K.

The parameter `Al` has the same options as for `UnitGroup`.

```
MergeUnits(K, a)
MergeUnits(O, a)
```

Verbose	`UnitGroup`	*Maximum :* 6

For an order O or a number field with maximal order O and a unit $a \in O$, add the unit to the already known subgroup of U_O that is stored in O. Returns `true` if and only if the rank of the currently known unit group of O or K increases when a is merged with it.

```
UnitRank(O)
UnitRank(K)
```

Return the unit rank of the ring of integers O of a number field K.

Example H37E21

In our field defined by $x^4 - 420 * x^2 + 40000$, we obtain the class and unit groups as follows.

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> C := ClassGroup(K);
> C;
Abelian Group of order 1
> U := UnitGroup(K);
> U;
Abelian Group isomorphic to Z/2 + Z + Z + Z
Defined on 4 generators
Relations:
      2*U.1 = 0
> T := TorsionUnitGroup(K);
> T;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*T.1 = 0
```

`IsExceptionalUnit(u)`

An element x of an order O is an exceptional unit if both x and $x - 1$ are units in O. This function returns `true` if and only if the order element u is an exceptional unit.

`ExceptionalUnitOrbit(u)`

If u is an exceptional unit of an order O, then all of the units $u_1 = u$, $u_2 = \frac{1}{u}$, $u_3 = 1 - u$, $u_4 = \frac{1}{1-u}$, $u_5 = \frac{u-1}{u}$, $u_6 = \frac{u}{u-1}$ are exceptional. The set $\Omega(u)$ formed by $u_1, \ldots, u_6$ is called the *orbit* of u. Usually it will have 6 elements. This function returns a sequence containing the elements of $\Omega(u)$.

`ExceptionalUnits(O)`

Verbose `UnitEq` *Maximum :* 5

This function returns a sequence S of units of the order O such that any exceptional unit u of O is either in S or is in the orbit of some element of S.

37.8 Solving Equations

Magma can solve norm, Thue, index form and unit equations.

37.8.1 Norm Equations

Norm equations in the context of number fields occur in many applications. While Magma contains efficient algorithms to solve norm equations it is important to understand the difference between the various types of norm equations that occur. Given some element θ in a number field k together with a finite extension K/k, there are two different types of norm equations attached to this data:

- Diophantine norm equations, that is norm equations where a solution $x \in K$ is restricted to a particular order (or any additive subgroup), and
- field theoretic norm equations where any element in $x \in K$ with $N(x) = \theta$ is a solution.

While in the first case the number of *different* (up to equivalence) solutions is finite, no such restriction holds in the field case. On the other hand, the field case often allows to prove the existence or non-existence of solutions quickly, while no efficient tests exist for the Diophantine case. So it is not surprising that different methods are applied for the different cases. We will discuss the differences with the individual intrinsics.

`NormEquation(O, m)`

`All`	BoolElt	*Default :* `true`
`Solutions`	RngIntElt	*Default :* All
`Exact`	BoolElt	*Default :* `false`
`Ineq`	BoolElt	*Default :* `false`

Given an order O and an element m of the ground ring of O which can be a positive integer or an element of a suborder, this intrinsic solves a Diophantine norm equation.

This function returns a boolean indicating whether an element $\alpha \in O$ exists such that $\mathrm{N_{F/L}}(\alpha)$, the norm of α with respect to the subfield L of F (the field of fractions of O), equals m, and if so, a sequence of length at most `Solutions` of solutions α.

The parameter `Exact` may be used to indicate whether an exact solution is required (with `Exact := true`) or whether a solution up to a torsion unit suffices.

The maximal number of required solutions can be indicated with the `Solutions` parameter, but setting `All := true` will override this and the search will find all solutions.

If the order is absolute, then the parameter `Ineq` may be set to true. If so, all solutions x with $|N(x)| <= m$ will be found using a variation of Fincke's ellipsoid method ([Fin84, PZ89]).

Depending on whether the order is absolute maximal, absolute or (simple) relative, different algorithms are used.

If the order is an absolute maximal order, Magma will, in a first step, enumerate all integral ideals having the required norm (up to sign). Next, all the ideals are tested for principality using the class group based method. If `Exact := true`, then a third step is added: we try to find a unit in O of norm -1. This unit is used to sign adjust the solution(s). If there is no such unit, we drop all solutions of the wrong sign.

If the order is absolute, but not maximal, the norm equation is first solved in the maximal order using the above outlined method. In a second step, a complete set of representatives for the unit group of the maximal order modulo the units of O is computed and Magma attempts to combine solutions in the maximal order with those representatives to get solutions in O.

If `Solutions` is set, the search stops after the required number of solutions is found.

In case the order is of type `RngOrd` and in some imaginary quadratic field, the norm function is a positive definite quadratic form, thus algorithms based on that property are used. In case the right hand side m equals ± 1, lattice based methods are applied.

If `Ineq` is `true`, which is only supported for absolute fields, lattice enumeration techniques ([Fin84, PZ89]) based on Fincke's ellipsoid method are used.

If the order is (simply) relative different algorithms are implemented, depending on the number of solutions sought. However, common to all of them is that they (partially) work in the `AbsoluteOrder` of O.

If O is a relative maximal order and if we only want to find 1 solution (or to prove that there is none), Magma first looks for (integral) solutions in the field using an S-unit based approach as outlined in `NormEquation`. This step gives an affine subspace of the S-unit group that contains all integral solutions of our equation. In a second step, a simplex based technique is used to find totally positive elements in the subspace.

In `All` is given or `Solutions` is > 1, then lattice based methods are used ([Fie97, Jur93, FJP97]).

`NormEquation(F, m)`

`Primes`	ESEQ OF PRIME IDEALS	*Default :* []
`Nice`	BOOLELT	*Default :* `true`

Given a field F and an element m of the base field of F, this function returns a boolean indicating whether an element $\alpha \in F$ exists such that $\mathrm{N}_{\mathrm{F/L}}(\alpha)$, the norm of α with respect to the base field L of F equals m, and if so, a sequence of length 1 of solutions α.

The field theoretic norm equations are all solved using S-units. Before discussing some details, we outline the method.

- Determine a set S of prime ideals. We try to obtain a solution as a S-unit for this set S.
- Compute a basis for the S-units
- Compute the action of the norm-map
- Obtain a solution as a preimage.

In general, no effective method is known for the first step. If the field is relative normal however, it is known that is S generates the class group of F and if m is a S-unit, then S is large enough (*suitable* in ([Coh00, 7.5]) [Fie97, Sim02, Gar80]. Thus to find S we have to compute the class group of F. If a (conditional) class group is already known, it is used, otherwise an *unconditional* class group is computed. The initial set S consists of all prime ideals occurring in the decomposition of $m\mathbf{Z}_F$. Note that this step includes the factorisation of m and thus can take a long time is m is large.

Next, we determine a basis for the S-unit group and the action of the norm on it. This give the norm map as a map on the S-unit group as an abstract abelian group.

Finally, the right hand side m is represented as an element of the S-unit group and a solution is then obtained as a preimage under the norm map.

If `Nice` is true, then MAGMA attempts to find a smaller solution by applying a LLL reduction to the original solution.

If `Primes` is give it must contain a list of prime ideals of L. Together with the primes dividing m it is used to form the set S bypassing the computation of an unconditional class group in this step. If L is not normal this can be used to guarantee that S is large enough. Note that the class group computation is still performed when the S-units are computed. Since the correctness of the S-unit group (we need only p-maximality for all primes dividing the (relative) degree of L) can be verified independently of the correctness of the class group, this can be used to derive provable results in cases where the class group cannot be computed unconditionally.

By default, the `MaximalOrder(L)` is used to compute the class group. If the attribute `NeqOrder` is set on L it must contain a maximal order of L. If present, this order will be used for all the subsequent computations.

```
NormEquation(m, N)
```

`Raw`	BOOLELT	*Default :* `false`
`Primes`	ESEQ OF PRIME IDEALS	*Default :* []

Let N be a map on the multiplicative group of some number field. Formally N may also be defined on the maximal order of the field. This intrinsic tries to find a pre-image for m under N.

This function works by realising N as a endomorphism of S-units for a suitable set S.

If N is a relative norm and if L is (absolutely) normal then the set S as computed for the field theoretic norm equation is guaranteed to be large enough to find a solution if it exists. Note: this condition is not checked.

If `Primes` is given it will be supplemented by the primes dividing m and then used as the set S.

If `Raw` is given, the solution is returned as an unevaluated power product. See the example for details.

The main use of this function is for Galois theoretical constructions where the subfields are defined as fields fixed by certain automorphisms. In this situation the norm function can be realised as the product over the fixed group. It is therefore not necessary to compute a (very messy) relative representation of the field.

`IntegralNormEquation(a, N, O)`

`Nice` BoolElt *Default :* `true`

For a an integer or a unit in some number field, N being a multiplicative function on some number field k which is the field of fractions of the order O, try to find a unit in O that is the preimage of a under N. In particular, N restricted to O must be an endomorphism of O. If `Nice` is `true`, the solution will be size-reduced. In particular when the conductor of O in the maximal order of k is large, and therefore the unit index $(Z_k)^* : O^*$ is large as well, this function is much more efficient than the lattice based approach above.

`SimNEQ(K, e, f)`

`S` [RngOrdIdl] *Default :* `false`

`HasSolution` BoolElt *Default :* `false`

For a number field K and subfield elements $e \in k_1$ and $f \in k_2$, try to find a solution to the simultaneous norm equations $N_{K/k_1}(x) = e$ and $N_{K/k_2}(x) = f$. The algorithm proceeds by first guessing a likely set S of prime ideals that will support a solution - it is exists. Initially S will contain all ramified primes in K, the support of e and f and enough primes to generate the class group of K. In case K is normal over Q this set is large enough to support a solution if there is a solution at all. For arbitrary fields that is most likely not the case. However, if `S` is passed in as a parameter then the set used internally will contain at least this set. If `HasSolution` is `true`, Magma will add primes to S until a solution has been found. This is useful in situations where for some theoretical reason it is known that there has to be a solution.

Example H37E22

We try to solve $N(x) = 3$ in some relative extension: (Note that since the larger field is a quadratic extension, the second call tells us that there is no integral element with norm 3)

```
> x := PolynomialRing(Integers()).1;
> O := MaximalOrder(NumberField([x^2-229, x^2-2]));
> NormEquation(O, 3);
false
> NormEquation(FieldOfFractions(O), 3);
true [
  5/1*$.1*O.1 + 2/3*$.1*O.2
]
```

Next we solve the same equation but come from a different angle, we will define the norm map as an element of the group ring and, instead of explicitly computing a relative extension, work instead with the implicit fix-field.

```
> K := AbsoluteField(FieldOfFractions(O));
> t := K!NumberField(O).2;
> t^2;
2/1*K.1
> A, _, mA := AutomorphismGroup(K);
> F := sub<A | [ x : x in A | mA(x)(t) eq t]>;
> N := map<K -> K | x:-> &* [ mA(y)(x) : y in F]>;
> NormEquation(3, N);
true [
    5/1*K.1 + 2/3*K.3
]
```

Finally, to show the effect of `Raw`:

```
> f, s, base := NormEquation(3, N:Raw);
> s;
[
    (0 -1 0 -2 0 1 0 0 0 -1 0 1 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 1 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)
]
> z := PowerProduct(base, s[1]);
> z;
5/1*K.1 + 2/3*K.3
> N(z);
3/1*K.1;
```

37.8.2 Thue Equations

Thue equations are Diophantine equations of the form $f(x, y) = k$, where k is some integer constant and f is a homogeneous polynomial in two variables. Methods for computing all solutions to such equations are known, although the search space may be larger than is practical. To work with such equations in Magma a Thue object (category `Thue`) must be created to store information related to the computations. To solve Thue equations the reduction of Bilu and Hanrot ([BH96]) is used.

`Thue(f)`

Given a polynomial f of degree at least 2 over the integers, this function returns the 'Thue object' corresponding to f; such objects are used by the functions for solving Thue equations. They are printed as the homogeneous version of f.

`Thue(O)`

Given an order O with $\mathbf{Z}$ as its coefficient ring, this function returns the Thue object corresponding to the defining polynomial of O.

`Evaluate(t, a, b)`

`Evaluate(t, S)`

Given a Thue object t and integers a, b, this function returns the evaluation of the homogeneous polynomial f involved in t at (a, b), that is $f(a, b)$. The second form takes as argument the sequence $[a, b]$ instead. This can be convenient if checking the results from an inexact solution.

`Solutions(t, a)`

`Exact`	BoolElt	*Default :* `true`
Verbose	`ThueEq`	*Maximum :* 5

Given a Thue object t and an integer a, this function returns a sequence consisting of all sequences of two integers $[x, y]$ which solve the equation $f(x, y) = a$, where f is the (homogeneous form of) the Thue equation associated with t. If the optional parameter `Exact` is set to `false` then solutions to $f(x, y) = -a$ will also be found.

Example H37E23

A use of thue equations is shown.

```
> R<x> := PolynomialRing(Integers());
> f := x^3 + x + 1;
> T := Thue(f);
> T;
Thue object with form:  X^3 + X Y^2 + Y^3
> Evaluate(T, 3, 2);
47
> Solutions(T, 4);
[]
> Solutions(T, 7);
[]
> Solutions(T, 47);
[
    [ -1, 4 ],
    [ 3, 2 ]
]
> S := Solutions(T, -47 : Exact := false);
> S;
[
    [ -3, -2 ],
    [ -1, 4 ],
    [ 1, -4 ],
    [ 3, 2 ]
]
> [Evaluate(T, s) : s in S];
[ -47, 47, -47, 47 ]
```

37.8.3 Unit Equations

Unit equations are equations of the form

$$a\epsilon + b\eta = c$$

where a, b and c are some algebraic numbers and ϵ and η are unknown units in the same field.

```
UnitEquation(a, b, c)
```

Verbose UnitEq *Maximum :* 5

Return the sequence of 1×2 matrices $(e1, e2)$ such that $ae_1 + be_2 = c$ for number field elements a, b and c, where e_1 and e_2 are units in the maximal order. The algorithm uses Wildanger's method ([Wil97, Wil00]).

Example H37E24

Usage of `UnitEquation` is shown.

```
> R<x> := PolynomialRing(Integers());
> K<a> := NumberField(x^7 - x^6 + 8*x^5 + 2);
> UnitEquation(a^7, 4*a^2 + a^80, a^7 + a^80 + 4*a^2);
[
    [[1, 0, 0, 0, 0, 0, 0] [1, 0, 0, 0, 0, 0, 0]]
]
```

37.8.4 Index Form Equations

Given an absolute number field K with some order O, index form equations are equations of the form $(O : Z[\alpha]) = k$ where k is some positive integer.

In particular, if $k = 1$ this function will find "all" integral power bases.

In this context "all" means up to equivalence, where two solutions α and β are equivalent iff $\alpha = \pm\beta + r$ for some integer r.

If the field degree is larger than 4, the field must be normal and an integral power basis must already be known.

The implementation follows [Wil97, Wil00] for large degree equations, [GPP93, GPP96] for quartics and [GS89] for cubic fields.

```
IndexFormEquation(O, k)
```

Verbose IndexFormEquation *Maximum :* 5

Given an absolute order O, this function will find "all" (up to equivalence) solutions $\alpha \in OP$ to $(O : Z[\alpha]) = k$.

Example H37E25

We try to compute all integral power bases of the field defined by a zero of $x^4 - 14x^3 + 14x^2 - 14x + 14$:

```
> x := PolynomialRing(Integers()).1;
> O := MaximalOrder(x^4-14*x^3+14*x^2-14*x+14);
> IndexFormEquation(O, 1);
[
    [0, 1, 0, 0]
    [0, 1, -13, 1],
    [0, 1, -1, 1],
]
> [ MinimalPolynomial(x) :x in $1 ];
[
    x^4 - 14*x^3 + 14*x^2 - 14*x + 14,
    x^4 - 28*x^3 + 56*x^2 + 3962*x - 28014,
    x^4 - 2044*x^3 + 6608*x^2 - 7126*x + 2562
]
> [ Discriminant(x) : x in $1 ] ;
[ -80240048, -80240048, -80240048 ]
> Discriminant(O);
-80240048
```

37.9 Ideals and Quotients

Ideals of orders are of two types. All ideals are fractional and inherit from type `RngOrdFracIdl`. Integral ideals can have (the sub–)type `RngOrdIdl`. Some functions only apply to integral ideals and only ideals with the integral type can be prime. An ideal with fractional type but trivial denominator can be converted to have the integral type and any integral ideal can be converted to fractional type.

Ideals can be taken of orders over **Z** and orders defined over a maximal order. A few functions are not implemented for the latter.

Where an element or elements are returned from a function the elements are usually in the field of fractions if the ideal has the fractional type and in the order if it has integral type.

37.9.1 Creation of Ideals in Orders

The general ideal constructor can be used to create ideals in orders of algebraic fields, as described below. Since ideals in orders are allowed to be *fractional ideals*, algebraic field elements are allowed as generators. Ideals can also be created how they are written on paper: as an element multiplied by an order.

```
x * O
O * x
```

Create the ideal $x * O$ for element x and order O. If the ideal is integral it will be returned with the integral type.

```
F !! I
O !! I
```

Make the ideal I either fractional (first case where F is a field of fractions compatible with the order of I) or integral (second case where O is an order compatible with the order of I).

```
ideal< O | a1, a2, ... , am >
ideal< O | x >
ideal< O | M, d >
ideal< O | M, I1, ..., In >
```

Given an order O, as well as anything that can be used to produce a sequence of elements of the field of fractions of O return the ideal generated by those elements. If the ideal is integral then it will be returned with the integral type.

A single integer may be given in which case the principal ideal it generates will be returned. A matrix or a module over an order (ModDed) (or a matrix and ideals which the module could be created from) can be supplied as the basis for the resulting ideal. An optional second argument is a denominator given as an integer, (except when ideals are given).

Example H37E26

We give an example of the creation of an ideal generated by an element from an order.

```
> R<x> := PolynomialRing(Integers());
> f := x^4-420*x^2+40000;
> K<y> := NumberField(f);
> E := EquationOrder(K);
> O := MaximalOrder(K);
> elt := O ! (y^2/40+y/4);
> elt in E;
false
> I := ideal< O | elt >;
> I;
Principal Ideal of O
```

```
Generator:
    [0, 0, 1, 0]
> FieldOfFractions(O)!!I;
Principal Ideal of O
Generator:
    O.3
> O!!$1 eq I;
true
```

37.9.2 Invariants

Some information describing an ideal can be retrieved.

`Order(I)`

The order O which the ideal I is of.

`Denominator(I)`

The denominator of the fractional ideal I. This is the smallest positive integer d such that $d * I$ is an integral ideal.

`PrimitiveElement(I)`
`UniformizingElement(P)`

A primitive element of an ideal I is an element a which is in I but not in the square of I. This function returns such an element a. `UniformizingElement` returns the primitive element of a prime ideal.

`Index(O, I)`

The index of the integral ideal I, when viewed as a submodule of the order O. This is the same as the cardinality of the finite quotient ring O/I.

`Norm(I)`

The norm of the fractional ideal I. This returns the index of the ideal if the ideal is integral, and is defined on fractional ideals by multiplicativity so that the norm of I^{-1} equals the reciprocal of the norm of I.

`MinimalInteger(I)`

Given an ideal I of an order in some number field, the function returns the least positive integer contained in the ideal.

`Minimum(I)`

Given an ideal I, this function returns the least positive integer m if the ideal is integral or the least positive rational r if is fractional contained in I.

`AbsoluteNorm(I)`

The absolute norm of the fractional ideal I. This returns the index of the ideal if the ideal is integral, and is defined on fractional ideals by multiplicativity so that the norm of I^{-1} equals the reciprocal of the norm of I.

`CoefficientHeight(I)`

`CoefficientHeight(I)`

For an ideal I the coefficient height is defined to be the maximum integer occurring in the current representation of the ideal: If the ideal is given via two elements, this will be the maximal coefficient height of the generators, otherwise the maximal entry of the basis matrix.

`CoefficientLength(I)`

`CoefficientLength(I)`

For an ideal I the coefficient length is defined to be the size of the current representation: If the ideal is given via two elements, this will be the sum of the coefficient lengths of the generators, otherwise the sum of the entries of the basis matrix.

`RamificationIndex(I, p)`

`RamificationDegree(I, p)`

For a prime ideal I of an order O such that $p \in I$ returns the maximal exponent e such that I^e divides the principal ideal pO. If p is not given it is taken to be the minimal integer of I.

`RamificationDegree(I)`

`RamificationIndex(I)`

Computes the relative ramification index of the prime ideal I over the coefficient ring. To be more precise: Let I be an prime ideal of some order O with coeffcient ring o. Then $I \cap o$ is an prime ideal p in o. The ramification index $e = e(I|p)$ is the maximal exponent e such that I^e divides pO.

`ResidueClassField(O, I)`

`ResidueClassField(I)`

If I is a prime ideal of O, this function returns the finite field F isomorphic to O/I and the map $O \to F$.

`Degree(I)`

`InertiaDegree(I)`

Given a prime ideal I this function returns the relative degree f of the residue class field of the ideal I. To be more precise: Let I be a prime ideal in some order O with coefficient ring o. Then $p := o \cap I$ is a prime ideal in o and the residue class field O/I is a finite extension of degree $f = f(I|p)$ of the residue class field o/p.

`Valuation(I, p)`

Given an ideal I and a prime ideal p in an order O, returns the valuation $v_p(I)$ of I at p, that is, the number of factors p in the prime ideal decomposition of I. Note that, since the ideal I is allowed to be a fractional ideal, the returned value may be a negative integer.

`Content(I)`

The content of the ideal I, i.e. the maximal ideal of the base ring dividing I.

Example H37E27

The retrieval of some properties of an ideal is illustrated.

```
> R<x> := PolynomialRing(Integers());
> M := MaximalOrder(x^5 + 4*x^4 - x^3 + 7*x^2 - 1);
> R<x> := PolynomialRing(M);
> O := MaximalOrder(x^3 - 2);
> M;
Maximal Equation Order with defining polynomial x^5 + 4*x^4 - x^3 + 7*x^2 - 1
over Z
> O;
Maximal Order of Equation Order with defining polynomial x^3 - [2, 0, 0, 0, 0]
over M
> I := 19/43*M.4*O.3*O;
> I;
Fractional Principal Ideal of O
Generator:
    19/43*M.4*O.3
> Order(I);
Maximal Order of Equation Order with defining polynomial x^3 - [2, 0, 0, 0, 0]
over M
> Denominator(I);
86
> Denominator(O.3);
2
> PrimitiveElement(I);
19/43*M.4*O.3
> Norm(I);
Fractional Ideal of M
Basis:
[6859    0    0    0    0]
[   0 6859    0    0    0]
[   0    0 6859    0    0]
[   0    0    0 6859    0]
[   0    0    0    0 6859]
Denominator: 159014
> Minimum(I);
19/43
```

```
> p := Factorization(3*M.2*O)[1][1];
> Valuation(I, p);
0
```

37.9.3 Basis Representation

The basis of an ideal can be computed as well as related matrices.

`Basis(I)`

`Basis(I, R)`

Given an ideal I of an order O of the algebraic field F, this function returns a basis for I as a sequence of elements of F (if fractional), O (if integral) or the ring R if given.

`BasisMatrix(I)`

Returns the basis matrix for the ideal I of the order O. The basis matrix consists of the elements of a basis for the ideal written as rows of rational coefficients with respect to the basis of O. The entries of the matrix are elements of **Z** for an integral ideal of an order over **Z** only or the field of fractions of the coefficient ring of O.

`TransformationMatrix(I)`

Returns the transformation matrix for the ideal I of the order O, as well as a denominator. The transformation matrix consists of the elements of a basis for the ideal written as rows of coefficients with respect to the basis of the order O. The entries of the matrix are elements of **Z** for an integral ideal of an order over **Z** or the field of fractions of the coefficient ring of O.

`CoefficientIdeals(I)`

The coefficient ideals of the ideal I in a relative extension. These are the ideals $\{A_i\}$ of the coefficient ring of the order of I such that for every element $e \in I$, $e = \sum_i a_i * b_i$ where $\{b_i\}$ is the basis returned for I and each $a_i \in A_i$.

Example H37E28

Continuing from the last example, the use of the basis functions for ideals is shown.

```
> Basis(I);
[
    19/43*M.4*O.3,
    19/86*M.4*O.1,
    19/43*M.4*O.2
]
> Basis(I, NumberField(O));
[
    19/86*$.1^3*$.1^2,
    19/86*$.1^3,
    19/86*$.1^3*$.1
]
> BasisMatrix(I);
[0 0 19/43*M.4]
[19/86*M.4 0 0]
[0 19/43*M.4 0]
> TransformationMatrix(I);
[M.1 0 0]
[0 M.1 0]
[0 0 M.1]
1
```

For relative extensions, a different method is available:

`Module(I)`

For an ideal I in some relative extension, return a Dedekind module over the coefficient ring with the "same" basis.

37.9.4 Two–Element Presentations

All ideals of maximal orders can be generated by one or two elements of the field of fractions of the order they are an ideal of.

`Generators(I)`

Given a (fractional) ideal I of O, return a sequence containing two elements that generate I as an ideal. The elements will be in the order iff the ideal is integral, otherwise they will come from the field of fractions of O.

`TwoElement(I)`

Given a (fractional) ideal I of O, return two elements of (the field of fractions of) O that generate I as an ideal.

`TwoElementNormal(I)`

Given an integral ideal I of O (a maximal order over $\mathbf{Z}$), return two elements of O that form a two-element normal presentation for I, as well as an integer g such that I is g-normal.

Example H37E29

The generators and two element presentation of I are compared.

```
> Generators(I);
[
    19/43*M.4*O.3
]
> TwoElement(I);
19/43*M.4*O.3
19/43*M.4*O.3
```

37.9.5 Predicates on Ideals

Ideals may be tested for various properties and whether given elements lie in the ideal.

`I eq J` `I ne J`

`x in I` `x notin I` `I subset J`

`IsIntegral(I)`

Returns `true` if and only if the fractional ideal I is integral.

`IsZero(I)`

Returns `true` if the ideal I is the zero ideal, `false` otherwise.

`IsOne(I)`

Returns `true` if the ideal I is the ideal generated by 1, `false` otherwise.

`IsPrime(I)`

Returns `true` if and only if the ideal I is a prime ideal, and `false` otherwise. If I has the integral type and is not prime, the function also returns a proper (ideal) divisor.

Currently, this function becomes very slow as the norm of the ideal becomes large.

`IsPrincipal(I)`

Verbose `ClassGroup` *Maximum :* 5

Returns `true` if the fractional ideal I of the order O is a principal ideal, otherwise `false`. If I is principal, a generator (as an element of the field of fractions of O) is also returned.

If it is known or easy to decide whether I is principal the function will return quickly. If it is necessary to compute the class group of O the function will be slower. If the generator is required this may cause the function to take longer as well.

`IsRamified(P)`

Returns `true` iff the ramification index of the prime ideal P is greater than 1.

`IsRamified(P, O)`

Returns `true` iff for any prime ideal Q lying above P in the order O, the ramification index of Q is greater than 1. P must be a prime ideal of the base ring of O or a prime number (if O is an absolute order).

`IsTotallyRamified(P)`

Returns `true` iff the ramification index of the prime ideal P equals the degree of its order over the base field.

`IsTotallyRamified(P, O)`

Returns `true` iff for any prime ideal Q lying above P in the order O, the ramification index of Q equals the field degree. P must be a prime ideal of the base ring of O or a prime number (if O is an absolute order).

`IsTotallyRamified(K)`

Returns `true` if all primes dividing the discriminant the maximal order of the algebraic field K are totally ramified over the coefficient field of K.

`IsTotallyRamified(O)`

Returns `true` if all primes dividing the discriminant of the maximal order O are totally ramified over the coefficient ring of O.

`IsWildlyRamified(P)`

Returns `true` iff the ramification index of the prime ideal P is a multiple of the characteristic of the residue class field of P.

`IsWildlyRamified(P, O)`

Returns `true` iff for any prime ideal Q of the order O lying above P, the ramification index $e(Q|P)$ is a multiple of the characteristic of the residue class field of Q. P must be a prime ideal of the base ring of O or a prime number (if O is an absolute order).

`IsTamelyRamified(P)`

Returns `true` iff the prime ideal P is not wildly ramified.

`IsTamelyRamified(P, O)`

Returns `true` iff the prime ideal or integer P is not wildly ramified in the order O.

`IsUnramified(P)`

Returns `true` iff the ramification index of the prime ideal P is 1.

`IsUnramified(P, O)`

Returns `true` iff for all prime ideals Q lying above P in the order O, the ramification index of Q is 1. P must be a prime ideal of the base ring of O or a prime number (if O is an absolute order).

`IsInert(P)`

Returns `true` iff the inertia degree of the prime ideal P is the field degree.

`IsInert(P, O)`

Returns `true` iff the prime integer or ideal P in the base ring of the order O is inert, i.e. PO is an unramified prime ideal of O.

`IsSplit(P)`

Returns `true` iff the prime ideal P is not the only prime ideal which lies above its intersection with the base ring of its order.

`IsSplit(P, O)`

Returns `true` iff at least two prime ideals in the order O lie above the prime integer or ideal P of the base ring of O.

`IsTotallySplit(P)`

Returns `true` iff there are as many prime ideals lying above the intersection of the prime ideal P with the base ring of its order as the degree of the order of P.

`IsTotallySplit(P, O)`

Returns `true` iff as many prime ideals in the order O lying above the prime integer or ideal P of the base ring of O as the degree of O.

37.9.6 Ideal Arithmetic

Ideals can be multiplied in several ways, divided and added. Powers of ideals, the least common multiple and the intersection of two ideals can also be calculated.

`I * J`

The product IJ of the (fractional) ideals I and J, generated by the products of elements in I and elements in J.

`x * I`

`I * x`

Given an element x of (or coercible into) a field of fractions F, and a (fractional) ideal I in the order of F, return the product of the ideal and the principal ideal generated by x.

`&*L`

The product of all ideals in the sequence or set L.

`I / J`

`I div J`

The quotient of the (fractional) ideals I and J of an maximal order O. This is the fractional ideal K of O with the property that $JK = I$.

`I div J`

For integral ideals I and J of some maximal order such that J divides I (or, equivalently, J is contained in I), return the integral ideal I/J.

`I / x`

Given an ideal I and an element x construct the fractional ideal I/x.

`I + J`

The sum of the (fractional) ideals I and J, generated by the sums of elements in I and elements in J.

`I ^ k`

The k-th power of the (fractional) ideal I (for an integer k). If I has integral type and k is negative the result will have fractional type.

`I eq J`

Tests if the ideals I and J are equal.

`I subset J`

Tests if the two ideals I and J of the same order are contained in each other. For invertible ideals this is equivalent to checking if J divides I,

`E in I`

Tests if the element E is actually in the ideal I. The element and the ideal have to be compatible, i.e., live in the same number field.

`LCM(I, J)`

`Lcm(I, J)`

`LeastCommonMultiple(I, J)`

Return the least common multiple of ideals I and J. They must both be of the same maximal order.

`GCD(I, J)`

`Gcd(I, J)`

`GreatestCommonDivisor(I, J)`

The greatest common divisor of the ideals I and J of some maximal order of an algebraic number field.

`Content(M)`

For a matrix M with entries in some number field k, compute the gcd of all elements as principal ideals in the maximal order of k.

`I meet J`

The intersection of the (fractional) ideals I and J. For ideal in the maximal order this is the same as the lcm.

`&meetS`

The intersection of all ideals I of the sequence or set S. For ideals in some maximal order this is the same as the lcm.

`I meet R`

`R meet I`

The intersection of the ideal I with the compatible ring R. If $R = \mathbf{Q}$ an error will occur since ideals of $\mathbf{Q}$ cannot be created. If such information is required use `Minimum` instead. An ideal of R is returned.

`a mod I`

A representative of the element a of an order O in the quotient O/I.

`InverseMod(E, M)`

`Modinv(E, M)`

An element y such that $y * E = 1 \bmod M$ where M is an integral ideal or an integer and E is an element of an order.

`ColonIdeal(I, J)`

`IdealQuotient(I, J)`

The colon ideal $[I : J]$ or (I/J) is defined as $\{x \in F : xJ \subseteq I\}$ where F is the field of fractions of the order the ideals I and J belong to.

For ideals of a maximal order (or in general for *invertible* ideals) this is equivalent to I/J, otherwise only `J*ColonIdeal(I, J) subset 1*Order(I)` holds.

`IntegralSplit(I)`

Given an ideal I, return an integral ideal J and a minimal positive integer d such that $I = J/d$.

37.9.7 Roots of Ideals

It is possible to ask for the k-th root of an ideal where k is a positive integer.

`Root(I, k)`

Find the kth root of the ideal I if it exists.

`IsPower(I, k)`

Return `true` if the ideal I is a kth power of an ideal and the ideal it is a power of otherwise `false`.

`SquareRoot(I)`

`Sqrt(I)`

Return the square root of the ideal I if I is square.

`IsSquare(I)`

Return `true` if the ideal I is the square of an ideal and the ideal it is a square of otherwise `false`.

37.9.8 Factorization and Primes

The factorization of an ideal into prime ideals and the divisors of an ideal can be determined.

`Decomposition(O, p)`

Verbose `IdealDecompose` *Maximum :* 5

Given an order O and a rational prime number p or a prime ideal p of the coefficient ring of O, return a sequence of tuples consisting of prime ideals and exponents, according to the decomposition of p in O.

`DecompositionType(O, p)`

Verbose `IdealDecompose` *Maximum :* 5

Given an order O and a rational prime number p or a prime ideal p of the coefficient ring of O, return the decomposition type, ie. a sequence of tuples consisting of the degree of the prime ideals and their ramification index, according to the decomposition of p in O.

`Factorization(I)`

`Factorisation(I)`

Verbose `IdealDecompose` *Maximum :* 5

Returns the prime ideal factorization of an ideal I in an order O, as a sequence of 2-tuples (prime ideal and integer exponent).

`Divisors(I)`

Return the ideals which divide the ideal I which must be of a maximal order.

`Support(I)`

For a non-zero ideal I of some maximal order, return the set of prime ideals p dividing I.

`Support(L)`

`GaloisStable`	BoolElt	*Default :* `false`
`CoprimeOnly`	BoolElt	*Default :* `false`
`UseBernstein`	BoolElt	*Default :* `false`

For a sequence L of ideals in some maximal order (or of number field elements representing principal ideals), return the set of prime ideals dividing at least one of the ideals. If `CoprimeOnly` is given, the set returned will not in general be containing prime ideals, but will satisfy the following:

Every ideal in L can be uniquely decomposed into a power product of ideals in the set returned

The set is minimal and closed under gcd, ie. for two elements in the set, their gcd will be one.

If the number field is normal and if `GaloisStable` is given, then the set returned will be closed under the action of the galois group. Depending on the (unknown) factorisation pattern of the ideals, taking the Galois action into account will in general refine the coprime factorisation.

In general, this function will first construct a coprime basis, and the factorise the result of this step.

If `UseBernstein` is given, then Dan Berstein's asymptoically fast algorithm ([Ber05], which runs in time essentially linear in $\#L$) is used.

`CoprimeBasis(L)`

`GaloisStable`	BoolElt	*Default :* `false`
`UseBernstein`	BoolElt	*Default :* `false`

Given a sequence L of ideals in some maximal order, a coprime basis C for L is constructed. That means

- every element in L has a unique representation as a power product with elements in C
- C is closed under gcd, the ideals in C are pairwise coprime.

If the field is normal and if `GaloisStable` is given, the input sequence is supplemented by the action of the automorphism group, thus potentially refining the coprime basis.

If `UseBernstein` is given then instead of the naive algorithm with quadratic complexity in $\#L$, an asymptotically fast, almost linear algorithm by Dan Berstein is used, [Ber05].

```
CoprimeBasisInsert(~L, I)
```

`GaloisStable`	BoolElt	*Default :* `false`
`UseBernstein`	BoolElt	*Default :* `false`

Given a coprime basis in the sequence L, enlarge it by the ideal I, ie. enlarge L in such a way that L stays a coprime basis but allows the decomposition of I as well. If the ideals are in a normal field and if `GaloisStable` is given, then in addition of I all its Galois conjugates are inserted as well. Furthermore, a fractional ideal is always decomposed into the numerator and denominator ideal, ach of which is inserted independently.

If `UseBernstein` is given then instead of the naive algorithm with quadratic complexity in $\#L$, an asymptotically fast, almost linear algorithm by Dan Berstein is used, [Ber05].

```
PowerProduct(B, E)
```

Given sequences B of ideals of some maximal order and E of integers, compute the ideal $\prod B[i]^{E[i]}$.

37.9.9 Other Ideal Operations

Various other functions can be applied to ideals. In addition to those listed, the completion of an order at a prime ideal can also be taken (see Completion on page 889 and Chapter 47).

```
ChineseRemainderTheorem(I1, I2, e1, e2)
ChineseRemainderTheorem(X, M)
CRT(I1, I2, e1, e2)
CRT(X, M)
```

Returns an element e of the order O such that $(e_1 - e)$ is in the ideal I_1 of O and $(e_2 - e)$ is in the ideal I_2. If a sequence of elements X and a sequence of ideals M is given then the element e will be such that $(X[i] - e)$ is in $M[i]$ for all i.

```
CRT(I1, L1, e1, L2)
ChineseRemainderTheorem(I1, L1, e1, L2)
```

Returns an element e of the order O such that $(e_1 - e)$ is in the ideal I_1 of O and the signs of the conjugates listed in $L1$ are the same as in $L2$.

$L1$, a sorted sequence of integers $0 < l_i <= r1$, is meant to be formal product of infinite places. The signs of the l_i'th conjugate of e will be the same as the sign of $L2[i]$.

```
Idempotents(I, J)
```

For coprime integral ideals I and J return `true` and elements $i \in I$ and $j \in J$ such that $i + j = 1$.

`CoprimeRepresentative(I, J)`

`MakeCoprime(I, J)`

Given two integral ideals I and J in the same maximal order, find an element q in the field of fractions of this order such that qI is coprime to J.

`ClassRepresentative(I)`

Let I be an ideal in the absolute maximal order O of the number field K. Further, assume that the class group of O has been computed. The class group calculation will have chosen a set of ideal class representatives. This function returns the representative ideal for the ideal class to which I belongs.

`Lattice(I)`

`MinkowskiLattice(I)`

Given an ideal I in an absolute order, returns the lattice determined by the real and complex embeddings of I.

`Different(I)`

The different of the (possibly fractional) ideal I of an order of an algebraic number field.

`Codifferent(I)`

The codifferent of the ideal I. This will be the inverse of the different of I if I is an ideal of a maximal order.

`SUnitGroup(I)`

`SUnitGroup(S)`

`Raw`	BoolElt	*Default :* `false`
Verbose	`ClassGroup`	*Maximum :* 5

An element mu of F, the field of fractions of the order of I, is an S-unit iff $v_p(mu) = 0$ for all p not in S. This function returns the group of S-units of the prime ideals given either as a sequence S of ideals or as a product ideal I. The map from the group into the order containing the ideals is also returned.

If `Raw` is `true` then a product representation of the units is returned instead. That means a sequence L of order elements is determined such that the S-unit group is contained in their multiplicative span. The S-unit group is represented as the $\mathbf{Z}$-module of the exponent vectors for L that give rise to some S-unit. Formally we obtain the S-unit group as an abstract abelian group A, a map from A into some large `RSpace` and the sequence L.

Example H37E30

First we compute the 3-units of $\mathbf{Q}(\sqrt{10})$.

```
> K := QuadraticField(10);
> M := MaximalOrder(K);
> U, mU := SUnitGroup(3*M);
> U;
Abelian Group isomorphic to Z/2 + Z + Z + Z
Defined on 4 generators
Relations:
    2*U.1 = 0
> mU;
Mapping from: GrpAb: U to Field of Fractions of M
> u := mU(U.3); u;
7/1*M.1 - 2/1*M.2
> Decomposition(u);
[
  <Prime Ideal of M
    Two element generators:
      3
      $.2 + 1, 2>
]
```

So u is indeed a 3 unit, as the factorization contains only prime ideals over 3. Next we do the same computation but using the `Raw` option:

```
> U, mU, base := SUnitGroup(3*M:Raw);
> mU;
Mapping from: GrpAb: U to Full RSpace of degree 14 over Integer Ring
> mU(U.3);
( 0  2  1  0  0  0  0  0  0  0 -2)
> PowerProduct(base, $1);
7/1*M.1 - 2/1*M.2
> base[2]^2 * base[3]^1 * base[11]^-2;
7/1*M.1 - 2/1*M.2
```

This representation is of particular importance for large degree fields or fields with large units. To illustrate this, consider the following field from the pari-mailing list:

```
> K := NumberField(Polynomial([ 13824, -3894, -1723, 5, 1291, 1 ]));
> L := LLL(MaximalOrder(K));
> C, mC := ClassGroup(L:Bound := 500);
> U, mU, base := SUnitGroup(1*L:Raw);
> logs := Matrix([Logs(x) : x in Eltseq(base)]);
> mU(U.3)*logs;
(2815256.998090806477937318458440358713392011019115562
    -636746.05251910981832558348350489744160309616673457
```

```
  -770882.4465262934206430757457152819150929093428O166)
```

As the logarithm of the absolute value of the real embeddings is of the order 10^6, we expect that a basis representation will have coefficients requiring roughly 10^6 digits. While it is feasible to compute them (using `PowerProduct`), this will take a long time.

`SUnitAction(SU, Act, S)`

`Base` SeqEnum[RngOrdElt] *Default :* []

Given a description of the S-unit group as computed by `SUnitGroup` and a (multiplicative) map of the underlying number field, this function computes the induced map on the abstract abelian group.

The argument S should be a sequence of ideals as in `SUnitGroup`. The argument SU should either be the map returned by `SUnitGroup(S)` as second return value - in which case `Base` is trivial (and not specified) or the second return value of `SUnitGroup(S:Raw)` in which case `Base` should equal the third computed value.

The argument Act must be any (multiplicative) function of the underlying number field or any order, that acts on the S-unit group.

On return, a endomorphism of the domain of SU is obtained.

`SUnitAction(SU, Act, S)`

`Base` SeqEnum[RngOrdElt] *Default :* []

Given a description of the S-unit group as computed by `SUnitGroup` and a sequence of (multiplicative) maps of the underlying number field, this function computes the induced maps on the abstract abelian group.

The argument S should be a sequence of ideals as in `SUnitGroup`. The argument SU should either be the map returned by `SUnitGroup(S)` as second return value - in which case `Base` is trivial (and not specified) or the second return value of `SUnitGroup(S:Raw)` in which case `Base` should equal the third computed value.

The argument Act must be a sequence of (multiplicative) functions of the underlying number field or any order, that acts on the S-unit group.

On return, a sequence of endomorphisms of the domain of SU is obtained.

`SUnitDiscLog(SU, x, S)`

`SUnitDiscLog(SU, L, S)`

`Base` SeqEnum[RngOrdElt] *Default :* []

Given a description of the S-unit group as computed by `SUnitGroup` and a (multiplicative) map of the underlying number field, this function computes the induced map on the abstract abelian group.

The argument S should be a sequence of ideals as in `SUnitGroup`. The argument SU should either be the map returned by `SUnitGroup(S)` as second return value - in which case `Base` is trivial (and not specified) or the second return value of `SUnitGroup(S:Raw)` in which case `Base` should equal the third computed value.

This function solves the discrete logarithm problem for the S-unit group and the algebraic number x. That is, an element in the abstract abelian group representing the S-unit group is computed which corresponds to x. If a list of algebraic numbers L is passed into this function, the discrete logarithm is computed for each of them.

Example H37E31

```
> M := MaximalOrder(Polynomial([ 25, 0, -30, 0, 1 ]));
> S := [ x[1] : x in Factorisation(30*M)];
> U, mU := SUnitGroup(S);
> L := Automorphisms(NumberField(M));
> s2 := SUnitAction(mU, L[2], S);
> s2;
Mapping from: GrpAb: U to GrpAb: U
> L[2](mU(U.2)) eq mU(s2(U.2));
```

Now the same in `Raw` representation:

```
> R, mR, Base := SUnitGroup(S:Raw);
> S2 := SUnitAction(mR, L[2], S:Base := Base);
> [S2(R.i) : i in [1..Ngens(R)]];
[
    R.1,
    R.1 - R.2,
    R.3,
    R.1 + R.3 - R.4,
    R.1 + R.5,
    R.1 + R.3 + R.7,
    R.1 - R.3 + R.6,
    R.8
]
```

If we combine `SUnitAction` with `SUnitDiscLog` we can solve norm equations:

```
> N := map<M -> M | x:-> L[1](x) * L[2](x)>;
> NR := SUnitAction(mR, N, S:Base := Base);
```

Now `NR` is the norm function with respect to the field fixed by `L[2]`.

```
> SUnitDiscLog(mR, FieldOfFractions(M)!5, S:Base := Base);
2*R.5
> $1 in Image(NR);
true
> $2 @@ NR;
R.2 + R.5
> PowerProduct(Base, mR($1));
-3/1*M.1 - 2/1*M.2 + M.4
> N($1);
[5, 0, 0, 0]
```

37.9.10 Quotient Rings

Quotients of orders defined over maximal orders and their integral ideals can be formed resulting in an object with type `RngOrdRes`. Elements of such orders can be created and elementary arithmetic and predicates may be applied to them.

37.9.10.1 Operations on Quotient Rings

The creation of quotient rings and the functions which may be applied to them are described.

```
quo< O | I >
```
```
quo< O | M >
```
```
quo< O | S >
```

Creates the quotient ring $Q = O/I$ of the order O. The right hand side of the constructor may contain an ideal or anything that the ideal constructor can create an ideal from.

```
UnitGroup(OQ)
```
```
MultiplicativeGroup(OQ)
```

Returns an abelian group and the map from the group into OQ. OQ must be a quotient of an absolute maximal order.

```
Modulus(OQ)
```

Return the denominator of the quotient ring OQ, i.e. I where $OQ = O/I$.

Example H37E32

Creation of quotient rings in shown. The orders are the same as for the ideal examples, however an integral ideal is now required.

```
> I := Denominator(I)*I;
> I;
Principal Ideal of O
Generator:
    38/1*M.4*O.3
> Basis(I);
[
    38/1*M.4*O.3,
    19/1*M.4*O.1,
    38/1*M.4*O.2
]
> Q := quo<Order(I) | I>;
> Q;
Quotient Ring of Principal Ideal of O
Generator:
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 38, 0]]
> Modulus(Q);
```

```
Principal Ideal of O
Generator:
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 38, 0]]
```

37.9.10.2 Elements of Quotients

Functions for elements of the quotient rings are predominantly arithmetic.

`OQ ! a`

Coerce the element a into the quotient OQ where a is anything that can be coerced into the order OQ is a quotient of.

`a mod I`

A canonical representative of the element a (belonging to an order O) in the quotient ring O/I.

`a * b` `a + b` `a - b` `a / b` `- a` `a ^ n`
`a eq b` `a ne b`

`IsZero(a)`

Returns `true` if and only if the quotient ring element a is the zero element of the quotient ring OQ.

`IsOne(a)`

Returns `true` if and only if the quotient ring element a is the one element of the quotient ring OQ.

`IsMinusOne(a)`

Returns `true` if and only if the quotient ring element a is the minus one element of the quotient ring OQ.

`IsUnit(a)`

Returns `true` if and only if the quotient ring element a has an inverse in the quotient ring OQ.

`Eltseq(a)`
`ElementToSequence(a)`

The coefficients of the quotient ring element a in the field of fractions of the coefficient ring of the order of the quotient ring containing a.

37.9.10.3 Reconstruction

Given an element e in some order O, known modulo some ideal I, a common problem in several algorithms is to recover e, ie. a unique minimal element $f \in O$ such that $e - f \in I$ and f is as "small" as possible. An equally common variation would be to ask for some field element f/d with d an integer, such that $f - de \in I$ and f as well as d are small.

In the case of O being the ring of integers and I a power of a prime (ideal), this is usually done by moving to the symmetric residue system in the first case and by rational reconstruction in the second. Here, we use techniques based on the LLL algorithm as described in [FF00].

Since the method is more complicated than in the case of the integer ring, one first has to create a "reconstruction environment" of type `RngOrdRecoEnv`, which is subsequently used to reconstruct any number of elements.

`ReconstructionEnvironment(p, k)`

`ReconstructionEnvironment(p, k)`

Given a (prime) ideal p and an exponent k, initialize the reconstruction process for the ideal $I = p^k$, that is, the object returned can be used to reconstruct elements from "approximations" modulo p^k.

`Reconstruct(x, R)`

`Reconstruct(x, R)`

`UseDenominator` BoolElt *Default :* `false`

Given an order element e, thought to be an approximation modulo p^k where p and k are stored in the reconstruction environment R, return the unique minimal f in the same order such that $e - g \in p^k$. Is `UseDenominator` is `true`, then a field element is computed, otherwise a ring element will be found.

`ChangePrecision(~ R, k)`

Change the ideal $I = p^l$ stored in R to p^k.

Example H37E33

We illustrate the use of the reconstruction environment to find roots of some polynomial over a number field. We will first compute the roots over some completion, up to some precision, then "list" the elements back from the completion into the starting order and finally use reconstruction to get the roots.

```
> f := Polynomial([1,1,1,1,1]);
> M := MaximalOrder(f);
> P := Decomposition(M, 11)[1][1]; P;
Prime Ideal of M
Two element generators:
    [11, 0, 0, 0]
    [2, 1, 0, 0]
> C, mC := Completion(M, P:Precision :=  10);
> fC := Polynomial([c@ mC : c in Eltseq(f)]);
> rt := Roots(fC); rt;
> R := ReconstructionEnvironment(P, 10);
> [Reconstruct((x[1]) @@ mC, R) : x in rt];
[
    M.4,
    M.3,
    -M.1 - M.2 - M.3 - M.4,
    M.2
]
> [ Evaluate(f, x) : x in $1];
[
    0,
    0,
    0,
    0
 ]
```

37.10 Places and Divisors

A place of a number field K is a class of absolute values (valuations) that induce the same topology on the field. By a famous theorem of Ostrowski, places of number fields are either finite, in which case they are in a one-to-one correspondence with the on-zero prime ideals of the maximal order, or infinite. The infinite places are identified with the embedding of K into **R** or with pairs of embeddings into **C**.

The group of divisors is formally the free group generated by the finite places and the **R**-vectorspace generated by the infinite ones.

For more information see Section 34.8.

37.10.1 Creation of Structures

`Places(K)`

`DivisorGroup(K)`

The set of places of the number field K and the group of divisors of K respectively.

37.10.2 Operations on Structures

`d1 eq d2` `p1 eq p2`

`NumberField(P)`

`NumberField(D)`

The number field for which P is the set of places or D is the group of divisors.

37.10.3 Creation of Elements

`Place(I)`

The place corresponding to prime ideal I.

`Decomposition(K, p)`

`Decomposition(K, I)`

A sequence of tuples of places and multiplicities. When a finite prime (integer) p is given, the places and multiplicities correspond to the decomposition of p in the maximal order of K. When the infinite prime is given, a sequence of all infinite places is returned.

`Decomposition(K, p)`

For a number field K and a place p of the coefficient field of K, compute all places (and their multiplicity) that extend p. For finite places this is equivalent to the decomposition of the underlying prime ideal. The sequence returned will contain the places of K extending p and their ramification index.

For an infinite place p, this function will compute all extensions of p in K. In this case, the integer returned in the second component of the tuples will be 1 if p is complex or if p is real and extends to a real place and 2 otherwise.

`Decomposition(m, p)`

`Decomposition(m, p)`

For an extension K/k of number fields (where k can be Q as well), given by the embedding map $m : k \to K$, decompose the place p of k in the larger field. In case $k = Q$, the place is given as either a prime number or zero to indicate the infinite place. The sequence returned contains pairs where the first component is a place above p via m and the second is the ramification index.

`InfinitePlaces(K)`

`InfinitePlaces(O)`

A sequence containing all the infinite places of the number field K or the order O is returned.

`Divisor(pl)`

The divisor $1 * pl$ for a place pl.

`Divisor(I)`

The divisor which is the linear combination of the places corresponding to the factorization of the ideal I and the exponents of that factorization.

`Divisor(x)`

The principal divisor xO where O is the maximal order of the underlying number field of which x is an element. In particular, this computes a finite divisor.

`RealPlaces(K)`

For a number field K all sequence containing all real (infinite) places is computed. For an absolute field this are precisely the embeddings into R coming from the real roots of the defining polynomial.

37.10.4 Arithmetic with Places and Divisors

Divisors and places can be added, negated, subtracted and multiplied and divided by integers.

`d1 + d2` `- d` `d1 - d2` `d * k` `d div k`

37.10.5 Other Functions for Places and Divisors

`Valuation(a, p)`

The valuation of the element a of a number field or order at the place p.

`Valuation(I, p)`

The valuation of the ideal I at the finite place p.

`Support(D)`

The support of the divisor D as a sequence of places and a sequence of the corresponding exponents.

`Ideal(D)`

The ideal corresponding to the finite part of the divisor D.

`Evaluate(x, p)`

`Evaluate(x, p)`

`Evaluate(x, p)`

The evaluation of the number field element x in the residue class field of the place p, ie. for a finite place p this corresponds to the image under the residue class field map for the underlying prime ideal. For infinite places, this returns the corresponding conjugate, ie. a real or complex number.

`RealEmbeddings(a)`

`RealEmbeddings(a)`

The sequence of real embeddings of the algebraic number a is computed, ie. a is evaluated at all real places of the number field.

`RealSigns(a)`

`RealSigns(a)`

A sequence containing ±1 depending on whether the evaluation of the number field element a at the corresponding real place is positive of negative.

`IsReal(p)`

For an infinite place p, returns `true` if the corresponding embedding is real, ie. if `Evaluate` at p will give real results.

`IsComplex(p)`

For an infinite place p, return `true` if the corresponding embedding is complex, ie. if `Evaluate` at p will generally yield complex results.

`IsFinite(p)`

For a place p of a number fiel, return if the place is finite, ie. if it corresponds to a prime ideal.

`IsInfinite(p)`

For a place p of a number field return if the place is infinite, ie. if it corresponds to an embedding of the number field into the real or complex numbers. If the place is infinite, the index of the embedding it corresponds to is returned as well.

`Extends(P, p)`

For two places P of K and p of k where K is an extension of k, check whether P extends p. For finite places, this is equivalent to checking if the prime ideal corresponding to P dives, in the maximal order of K the prime ideal of p. For infinite places `true` implies that for elements of k, evaluation at P and p will give identical results.

`InertiaDegree(P)`

`Degree(P)`

For a place P of a number field, return the inertia degree of P. That is for a finite place, return the degree of the residue class field over it's prime field, for infinite places it is always 1.

`Degree(D)`

For a divisor D of a number field, the degree is the weighted sum of the degrees of the supporting places, the weights being the multiplicities.

`NumberField(P)`

For a place P or divisor D of a number field, return the underlying number field.

`ResidueClassField(P)`

For a place P of a number field, compute the residue class field of P. For a finite place this will be a finite field, namely the residue class field of the underlying prime ideal. For an infinite place, the residue class field will be the field of real or complex numbers.

`UniformizingElement(P)`

For a finite place P of a number field, return an element of valuation 1. This will be the uniformizing element of the underlying prime ideal as well.

`LocalDegree(P)`

The degree of the completion at the place P, ie. the product of the inertia degree times the ramification index.

`RamificationIndex(P)`

The ramification index of the place P. For infinite real places this is 1 and 2 for complex places.

`DecompositionGroup(P)`

For a place P of a normal number field, return the decomposition group as a subgroup of the (abstract) automorphism group.

37.11 Bibliography

[**Bai96**] Georg Baier. Zum Round 4 Algorithmus. Diplomarbeit, Technische Universität Berlin, 1996.
URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/baier.ps.gz.

[**Ber05**] Daniel J. Bernstein. Factoring into coprimes in essentially linear time. *J. of Algorithms*, 54(1):1–30, 2005.

[**BH96**] Yuri Bilu and Guillaume Hanrot. Solving Thue Equations of High Degree. *J. Number Th.*, 60:373–392, 1996.

[**Bia**] J.-F. Biasse. Number field sieve to compute Class groups.

[**Bj94**] Johannes A. Buchmann and Hendrik W. Lenstra jr. Approximating rings of integers in number fields. *J. Théor. Nombres Bordx.*, 6(2):221–260, 1994.

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.

[**Coh00**] Henri Cohen. *Advanced Topics in Computational Number Theory*. Springer, Berlin–Heidelberg–New York, 2000.

[**FF00**] Claus Fieker and Carsten Friedrichs. On reconstruction of algebraic numbers. In Bosma [Bos00], pages 285–296.

[Fie97] Claus Fieker. *Über relative Normgleichungen in algebraischen Zahlkörpern.* Dissertation, Technische Universität Berlin, 1997.
URL:http://www.math.tu-berlin.de/∼kant/publications/diss/diss_CF.ps.gz.

[Fin84] Ulrich Fincke. *Ein Ellipsoidverfahren zur Lösung von Normgleichungen in algebraischen Zahlkörpern.* Dissertation, Heinrich-Heine-Universität Düsseldorf, 1984.

[FJP97] C. Fieker, A. Jurk, and M. Pohst. On solving relative norm equations in algebraic number fields. *Math. Comput.*, 66(217):399–410, 1997.

[Fri97] Carsten Friedrichs. Berechnung relativer Ganzheitsbasen mit dem Round-2-Algorithmus. Diplomarbeit, Technische Universität Berlin, 1997.
URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/friedrichs.ps.gz.

[Fri00] Carsten Friedrichs. *Berechnung von Maximalordnungen über Dedekindringen.* Dissertation, Technische Universität Berlin, 2000.
URL:http://www.math.tu-berlin.de/∼kant/publications/diss/diss_fried.pdf.gz.

[Gar80] Dennis A. Garbanati. An Algorithm for finding an algebraic number whose norm is a given rational number. *J. reine angew. Math.*, 316:1–13, 1980.

[GPP93] Istvan Gaál, Attila Pethő, and Michael E. Pohst. On the resolution of index form equations in quartic number fields. *J. Symbolic Comp.*, 16:563–584, 1993.

[GPP96] Istvan Gaál, Attila Pethő, and Michael E. Pohst. Simultaneous representation of integers by a pair of ternary quadratic forms – With an application to index form equations in quartic number fields. *J. Number Th.*, 57:90–104, 1996.

[GS89] Istvan Gaál and Nicole Schulte. Computing all power integral bases of cubic fields. *Math. Comp.*, 53:689–696, 1989.

[Heß96] Florian Heß. Zur Klassengruppenberechnung in algebraischen Zahlkörpern. Diplomarbeit, Technische Universität Berlin, 1996.
URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/hess.ps.gz.

[Jur93] Andreas Jurk. *Über die Berechnung von Lösungen relativer Normgleichungen in algebraischen Zahlkörpern.* Dissertation, Heinrich-Heine-Universität Düsseldorf, 1993.

[KAN97] KANT Group. KANT V4. *J. Symbolic Comp.*, 24(3–4):267–383, 1997.

[KAN00] KANT Group. The Number Theory Package KANT/KASH.
URL:http://www.math.tu-berlin.de/∼kant, 2000.

[PK05] Sebastian Pauli and Jüren Klüners. Computing residue class rings and Picard groups of orders. *J. of Algebra*, 292:47–64, 2005.

[Poh93] M. Pohst. *Computational Algebraic Number Theory.* DMV Seminar Band 21. Birkhäuser Verlag, Basel - Boston - Berlin, 1993.

[PZ89] Michael E. Pohst and Hans Zassenhaus. *Algorithmic Algebraic Number Theory.* Encyclopaedia of mathematics and its applications. Cambridge University Press, Cambridge, 1989.

[Sim02] Denis Simon. Solving norm equations in relative number fields using S-units. *Math. Comput.*, 71(239):1287–1305, 2002.

[**Sut12**] Nicole Sutherland. Efficient Computation of Maximal Orders of Radical (including Kummer) Extensions. *Journal of Symbolic Computation*, 47:552–567, 2012.

[**Wil97**] Klaus Wildanger. *Über das Lösen von Einheiten- und Indexformgleichungen in algebraischen Zahlkörpern mit einer Anwendung auf die Bestimmung aller ganzen Punkte einer Mordellschen Kurve.* Dissertation, Technische Universität Berlin, 1997. URL:http://www.math.tu-berlin.de/~kant/publications/diss/KW_diss.ps.gz.

[**Wil00**] Klaus Wildanger. Über das Lösen von Einheiten- und Indexformgleichungen in algebraischen Zahlkörpern. (On the solution of units and index form equations in algebraic number fields). *J. Number Th.*, 2(82):188–224, 2000.

38 GALOIS THEORY OF NUMBER FIELDS

Chapter 38
GALOIS THEORY OF NUMBER FIELDS

The Galois theory of number fields deals with the group of automorphisms of a number field, the group of automorphisms of the normal closure of a number field and with the subfields of a given number field. While all three problems are, at least in theory, dealt with easily using the main theorems of Galois theory, they correspond to completely different and independent algorithmic problems.

The first task, that of computing automorphisms of normal extensions of **Q** (and of abelian extensions of number fields) can be thought of a special case of factorisation of polynomials over number fields: the automorphisms of a number field are in one-to-one correspondence with the roots of the defining equation in the field. However, the computation follows a different approach and is based on some combinatorial properties. It should be noted, though, that the algorithms only apply to normal fields; i.e., they cannot be used to find non-trivial automorphisms of non-normal fields!

The second task, namely that of computing the Galois group of the normal closure of a number field, is of course closely related to the problem of computing the Galois group of a polynomial. The method implemented in Magma allows the computation of Galois groups of polynomials (and number fields) of arbitrarily high degrees and is independent of the classification of transitive permutation groups. The result of the computation of a Galois group will be a permutation group acting on the roots of the (defining) polynomial, where the roots (or approximations of them) are explicitly computed in some suitable p-adic field; thus the splitting field is not (directly) part of the computation. The explicit action on the roots allows one, for example, to compute algebraic representations of arbitrary subfields of the splitting field, even the splitting field itself, provided the degree is not too large.

The last main task dealt with in this chapter is the computation of subfields of a number field. While of course this can be done using the main theorem of Galois theory (the correspondence between subgroups and subfields), the computation is completely independent; in fact, the computation of subfields is usually the first step in the computation of the Galois group. The algorithm used here is mainly combinatorical.

Finally, this chapter also deals with applications of the Galois theory:

- the computation of subfields and subfield towers of the splitting field
- solvability by radicals: if the Galois group of a polynomial is solvable, the roots of the polynomial can be represented by (iterated) radicals.
- basic Galois-cohomology; i.e., the action of the automorphisms on the ideal class group, the multiplicative group of the field and derived objects.

38.1 Automorphism Groups

Automorphisms of an algebraic field and the group they form can be calculated. Furthermore, field invariants that relate to the automorphism group can be determined.

`Automorphisms(F)`

`Abelian`	BoolElt	*Default :* `false`
Verbose	`AutomorphismGroup`	*Maximum :* 3

Given an algebraic field F, return the automorphisms of F as a sequence of maps. If the extension is known to be abelian, the parameter `Abelian` should be set to `true` in which case a much more efficient algorithm [Klü97, AK99] will be employed. If F is not a normal extension, the automorphisms are obtained by a variation of the polynomial factorisation algorithm.

`AutomorphismGroup(F)`

`Abelian`	BoolElt	*Default :* `false`
Verbose	`AutomorphismGroup`	*Maximum :* 3

Given an algebraic field F, that is either a simple normal extension of **Q** or simple abelian extension of **Q**, return the automorphism group G of K as a permutation group of degree n, where n is the degree of the extension. If the extension is known to be abelian, the parameter `Abelian` should be set to `true` in which case a much more efficient algorithm [Klü97, AK99] will be employed. If F is not a normal extension of **Q** an error will occur. In addition to returning G, the function also returns the power structure Aut of all automorphisms of F, and the transfer map ϕ from G into Aut.

Example H38E1

We consider the extension obtained by adjoining a root of the irreducible polynomial $x^4 - 4x^2 + 1$ to **Q**.

```
> Q := RationalField();
> R<x> := PolynomialRing(Q);
> K<w> := NumberField(x^4 - 4*x^2 + 1);
> A := Automorphisms(K);
> A;
[
    Mapping from: FldNum: K to FldNum: K,
    Mapping from: FldNum: K to FldNum: K,
    Mapping from: FldNum: K to FldNum: K,
    Mapping from: FldNum: K to FldNum: K
]
> for phi in A do phi(w); end for;
w
w^3 - 4*w
-w^3 + 4*w
```

```
-w
```

Taking the same field K we use instead the function `AutomorphismGroup`:

```
> G, Aut, tau := AutomorphismGroup(K);
> for x in G do tau(x)(w); end for;
w
w^3 - 4*w
-w^3 + 4*w
-w
```

`AutomorphismGroup(K, F)`

Computes the group of K automorphisms of F as a permutation group together with a list of all automorphisms and a map between the permutation group and explicit automorphisms of the field.

This function computes the automorphism group of the F over **Q** first.

`DecompositionGroup(p)`

For an ideal p of the maximal order of some absolute normal field F with group of automorphisms G, compute the decomposition group, i.e. the subgroup U of the automorphism group such that:

$$U := \{s \in G | s(p) = p\}$$

If F is not a normal extension of **Q** an error will occur.

`RamificationGroup(p, i)`

For an ideal p of the maximal M order of some absolute normal field F with group of automorphisms G, compute the i-th ramification group, i.e. the subgroup U of the automorphism group such that:

$$U := \{s \in G | s(x) - x \in p^{i+1} \text{for all } x \text{ in } M\}$$

If F is not a normal extension of **Q** an error will occur.

`RamificationGroup(p)`

This is just an abbreviation for `RamificationGroup(p, 1)`.

`InertiaGroup(p)`

This is just an abbreviation for `RamificationGroup(p, 0)`.

```
FixedField(K, U)
```

Given a normal field K over $\mathbf{Q}$ and a subgroup U of the `AutomorphismGroup(K)`, compute the subfield L that is fixed by U.

This function is inverse to `FixedGroup`.

If K is not a normal extension of $\mathbf{Q}$ an error will occur.

```
FixedField(K, S)
```

For an algebraic field K and a list S of automorphism of K, compute the maximal subfield of K fixed by S.

```
FixedGroup(K, L)
```

Given a normal field K over $\mathbf{Q}$ and a subfield L, compute the subgroup U of the `AutomorphismGroup(K)` that fixes L.

This function is inverse to `FixedField`.

If K is not a normal extension of $\mathbf{Q}$ an error will occur.

```
FixedGroup(K, L)
```

Given a normal field K over $\mathbf{Q}$ and a sequence of number field elements L, compute the subgroup U of the `AutomorphismGroup(K)` that fixes L.

If K is not a normal extension of $\mathbf{Q}$ an error will occur.

```
FixedGroup(K, a)
```

Given a normal field K over $\mathbf{Q}$ and a number field element a, compute the subgroup U of the `AutomorphismGroup(K)` that fixes a.

This function is inverse to `FixedField`.

If K is not a normal extension of $\mathbf{Q}$ an error will occur.

```
DecompositionField(p)
```

This is just an abbreviation for `FixedField(K, DecompositionGroup(p))` where K is the number field of the order of p.

```
RamificationField(p, i)
```

This is just an abbreviation for `FixedField(K, RamificationGroup(p, i))` where K is the number field of the order of p.

```
RamificationField(p)
```

This is just an abbreviation for `FixedField(K, RamificationGroup(p))` where K is the number field of the order of p.

```
InertiaField(p)
```

This is just an abbreviation for `FixedField(K, InertiaField(p))` where K is the number field of the order of p.

Example H38E2

We will demonstrate the various groups and fields. In order to do so, we first construct a non-trivial normal field.

```
> o := MaximalOrder(ext<Rationals()|>.1^4-3);
> os := MaximalOrder(SplittingField(NumberField(o)));
> P := Decomposition(os, 2)[1][1];
> G, M := RayClassGroup(P^3);
> G;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*G.1 = 0
```

Since G is cyclic and the module P invariant under the automorphisms of os, the class field corresponding to G will be normal over Q. It Galois group over Q will be an extension of D_4 by C_2.

```
> A := AbelianExtension(M);
> O := MaximalOrder(EquationOrder(A));
> Oa := AbsoluteOrder(O);
> Ka := NumberField(Oa);
> Gal, _, Map := AutomorphismGroup(Ka);
> Gal;
Permutation group Gal acting on a set of cardinality 16
Order = 16 = 2^4
    (1, 2, 7, 5)(3, 8, 6, 10)(4, 12, 14, 9)(11, 16, 13, 15)
    (1, 3, 7, 6)(2, 8, 5, 10)(4, 13, 14, 11)(9, 16, 12, 15)
    (1, 4)(2, 9)(3, 11)(5, 12)(6, 13)(7, 14)(8, 15)(10, 16)
```

Now, let us pick some ideals. The only interesting primes are the primes dividing the discriminant, which in this case will be the primes over 2 and 3.

```
> P2 := Decomposition(Oa, 2)[1][1];
> P3 := Decomposition(Oa, 3)[1][1];
```

First, the valuation of the different of Oa at $P2$ should be $\sum_{i=0}^{\infty}(\#G(P2,i)-1)$ where $G(P2,i)$ is the i-th ramification group.

```
> s := 0; i := 0;
> repeat
>   G := RamificationGroup(P2, i);
>   s +:= #G-1;
>   print i, "-th ramification group is of order ", #G;
>   i +:= 1;
> until #G eq 1;
0 -th ramification group is of order  8
1 -th ramification group is of order  8
2 -th ramification group is of order  2
3 -th ramification group is of order  2
```

```
4 -th ramification group is of order  2
5 -th ramification group is of order  2
6 -th ramification group is of order  1
> s;
18
> Valuation(Different(Oa), P2);
18
```

According to the theory, $P2$ should be totally ramified over the inertia field and unramified over Q:

```
> K2 := InertiaField(P2);
> M2 := MaximalOrder(K2);
> K2r := RelativeField(K2, Ka);
> M2r := MaximalOrder(K2r);
> p2 := M2 meet (MaximalOrder(K2r)!!P2);
> IsInert(p2);
true
> IsTotallyRamified(M2r!!P2);
true
```

Now we try the same for $P3$. Since 3 is split in Ka, we may consider an additional field: the decomposition field. It should be the maximal subfield if K such that 3 is neither inert ($f = 1$) nor ramified ($e = 1$), therefore 3 has to split totally.

```
> D3 := DecompositionField(P3);
> D3M := MaximalOrder(D3);
> IsTotallySplit(3, D3M);
true
```

The inertia field is the maximal subfield such that 3 is unramified. It has to be an extension of $D3$.

```
> I3 := InertiaField(P3);
> I3;
Number Field with defining polynomial $.1^4 +
    803463845096310571824l2*$.1^3 +
    2256835583037881432653115137736209396615693022*$.\
    1^2 + 27958180928554764690569897399558457365792916051 77\
    3809455107173769804*$.1 +
    2207787685682553980385342263526644079975418801375161428\
    411471043013257604817288336500609 94 over the Rational
Field
> Discriminant($1);
10700005925626216180895747020647047166414333000723923591882\
5782987341763807211711494516350753784471154461714734422 7643\
2140850348956694986629566940082522274866090780823540144 4104\
2932949364571465839467357930989372653299974549668957108 2958\
82869371250900344499670337698224 64
```

This (polynomial) discriminant is huge, in fact it is so large that we should avoid the factorisation. We already know the discriminant of Ka. The discriminant of $I3$ has to be a divisor - so we can use the `Discriminant` parameter to `MaximalOrder`: (We are going to need the `MaximalOrder` for the following embedding.)

```
> I3M := MaximalOrder(EquationOrder(I3):
> Discriminant := Discriminant(Oa));
> I3M := MaximalOrder(I3);
```

*D*3 should be a subfield of *I*3, so lets verify it:

```
> IsSubfield(D3, I3);
true Mapping from: FldNum: D3 to FldNum: I3
```

As a side-effect, Magma is now aware of the embedding and will use it. Without the `IsSubfield` call, the `RelativeField` function will fail.

```
> I3r := RelativeField(D3, I3);
> I3rM := MaximalOrder(I3r);
> K3r := RelativeField(D3, Ka);
> K3rM := MaximalOrder(K3r);
> IsInert(K3rM!!P3 meet D3M, I3rM);
true
```

The last step: verify that *P*3 is totally ramified over *I*3:

```
> K3r := RelativeField(I3, Ka);
> K3rM := MaximalOrder(K3r);
> IsTotallyRamified(K3rM!!P3 meet I3M, K3rM);
true
```

Using the decomposition group, we can get the splitting behaviour of any prime in any subfield of *Ka*.

```
> L := SubgroupLattice(Gal);
> [ IsNormal(Gal, L[x]) : x in [1..#L]];
[ true, true, true, true, false, false, false, false, true,
true, true, true, true, true, true, false, false, false,
false, true, true, true, true, true, true, true, true ]
> U := L[5];
> k := FixedField(Ka, U);
> kM := MaximalOrder(EquationOrder(k):
>   Discriminant := Discriminant(Oa));
> kM := MaximalOrder(k);
> Kr := RelativeField(k, Ka);
> KrM := MaximalOrder(Kr);
> P43 := Decomposition(Oa, 43)[1][1];
> V := DecompositionGroup(P43);
```

The splitting behaviour is determined by the double coset decomposition of *Gal* with respect to *U* and *V*:

```
> f, I := CosetAction(Gal, U);
> orbs := Orbits(f(V));
> reps := [];
> for o in orbs do
>   _, x := IsConjugate(I, 1, Rep(o));
>   Append(~reps, x @@ f);
> end for;
> reps;
[
    Id(G),
    (1, 2, 7, 5)(3, 8, 6, 10)(4, 12, 14, 9)(11, 16, 13, 15),
    (1, 7)(2, 5)(3, 6)(4, 14)(8, 10)(9, 12)(11, 13)(15, 16),
    (1, 8)(2, 6)(3, 5)(4, 15)(7, 10)(9, 13)(11, 12)(14, 16),
]
> #reps;
4
```

So there will be at least 4 prime ideals over 43 in k:

```
> L := [ ];
> for i in reps do
>   Append(~L, kM meet KrM !! Map(i)(P43));
> end for;
> [ IsPrime(x) : x in L];
[ true, true, true, true ]
> LL := Decomposition(kM, 43);#LL;
4
> [ Position(L, x[1]) : x in LL];
[ 4, 3, 1, 2 ]
```

`FrobeniusElement(K, p)`

Compute a Frobenius element at p in the Galois group of the Galois closure of K. This is a permutation on the roots of a polynomial defining K, which can be recovered as `DefiningPolynomial(A)` for any Artin representation A of K; the Frobenius element is well-defined up to conjugacy and modulo inertia.

Example H38E3

We take a polynomial whose Galois group is D_5 and compute Frobenius elements at $p = 2$ and $p = 5$. They in two different conjugacy classes of 5-cycles in the Galois group.

```
> load galpols;
> f:=PolynomialWithGaloisGroup(5,2);
> assert IsIsomorphic(GaloisGroup(f),DihedralGroup(5));
> K:=NumberField(f);
> FrobeniusElement(K,2);
(1, 5, 4, 3, 2)
> FrobeniusElement(K,5);
(1, 3, 5, 2, 4)
```

38.2 Galois Groups

Finding Galois groups (of normal closures) of algebraic fields is a hard problem, in general. All practical currently-used algorithms fall into two groups: The absolute resolvent method [SM85] and the method of Stauduhar [Sta73].

The Magma implementation is based on an extension of the method of Stauduhar [GK00, Gei03] and recent work by Klüners and Fieker [FK].

For polynomials over $\mathbf{Z}, \mathbf{Q}$, number fields and global function fields and irreducible polynomials over function fields over $\mathbf{Q}$, Magma is able to compute the Galois group without any a-priori restrictions on the degree. Note, however, that the running time and memory constraints can make computations in degree > 50 impossible, although computations in degree > 200 have been successful as well. In contrast to the absolute resolvent method, it also provides the explicit action on the roots of the polynomial f which generates the algebraic field. On demand, the older version which is restricted to a maximum degree of 23, is still available.

Roughly speaking, the method of Stauduhar traverses the subgroup lattice of transitive permutation groups of degree n from the symmetric group to the actual Galois group. This is done by using so-called relative resolvents. Resolvents are polynomials whose splitting fields are subfields of the splitting field of the given polynomial which are computed using approximations of the roots of the polynomial f.

If the field (or the field defined by a polynomial) has subfields (i.e. the Galois group is imprimitive) the current implementation changes the starting point of the algorithm in the subgroup lattice, to get as close as possible to the actual Galois group. This is done via computation of subfields of a stem field of f, that is the field extension of $\mathbf{Q}$ which we get by adjoining a root of f to $\mathbf{Q}$. Using this knowledge of the subfields, the Galois group is found as a subgroup of the intersection of suitable wreath products which may be easily computed. This intersection is a good starting point for the algorithm.

If the field (or the field defined by a polynomial) does not have subfields (i.e. the Galois group is primitive) we use a combination of the method of Stauduhar and the absolute resolvent method. The Frobenius automorphism of the underlying p-adic field or the complex conjugation, when using complex approximations of the roots of the polynomial f, already determines a subgroup of the Galois group, which is used to speed up computations in the primitive case.

```
GaloisGroup(f)
```

`Prime`	RngElt	*Default :*
`ShortOK`	BoolElt	*Default :* `false`
`Ring`	GaloisData	*Default :*
`NextPrime`	UserProgram	*Default :*
Verbose	`GaloisGroup`	*Maximum :* 5
Verbose	`Invariant`	*Maximum :* 3

Given a polynomial f over the integers, rationals, a number field or an order thereof, compute the Galois group of a splitting field for f, ie. determine the subgroup of the permutations of the roots of f in a splitting field that correspond to field automorphisms. The method applied here is a variant of Stauduhar's algorithm, but with no dependency on the explicit classification of transitive groups and thus no a-priori degree limitation. It must be stated though that this function does not return proven results, if such results are necessary, one needs to call `GaloisProof` afterwards.

The prime to use for splitting field computations can be given via the parameter `Prime`. The method of choosing of primes for splitting field computations can be given by the parameter `NextPrime`.

`GaloisGroup(K)`

`Prime`	RngElt	*Default :*
`ShortOK`	BoolElt	*Default :* `false`
`Ring`	GaloisData	*Default :*
`NextPrime`	UserProgram	*Default :*
`Current`	BoolElt	*Default :* `false`
`Subfields`	BoolElt	*Default :* `true`
`Old`	BoolElt	*Default :* `false`
`Type`	MonStgElt	*Default :* "p-Adic"
`Prec`	RngIntElt	*Default :* 20
`Time`	Rec	*Default :*
Verbose	`GaloisGroup`	*Maximum :* 5
Verbose	`Invariant`	*Maximum :* 3

Given a number field K, compute the Galois group of a normal closure of K. The group is returned as an abstract permutation group acting on the roots of the defining polynomial of k in a suitable splitting field. The method applied here is a variant of Stauduhar's algorithm, but with no dependency on the explicit classification of transitive groups and thus no a-priori degree limitation. It must be stated though that this function does not return proven results, if such results are necessary, one needs to call `GaloisProof` afterwards.

The prime to use for splitting field computations can be given via the parameter `Prime`. The method of choosing of primes for splitting field computations can be given by the parameter `NextPrime`.

`GaloisProof(f, S)`

`GaloisProof(K, S)`

Given the result of a (conditional) computation by `GaloisGroup` for either an irreducible polynomial over the integers or an absolute number field, try to find proofs for the conditional steps of the algorithm. The method employed here is to show that a suitable absolute resolvent polynomials has a factor of a specific degree that can be used to differentiate between the possible groups.

`GaloisRoot(f, i, S)`

`Prec`	RngIntElt	*Default :* 20
`Bound`	RngIntElt	*Default :* 0

Given a polynomial f and the result S of a computation of its Galois group, return the ith root in the ordering obtained form the Galois process to the given precision. The precision can be specified either directly by setting `Prec` to the desired p-adic precision or by giving a bound B in `Bound`. In the latter case the p-adic precision k will be calculated such that $p^k > B$.

`GaloisRoot(i, S)`

Given the result S of a computation of a Galois group and an integer i, compute the ith conjugate of the primitive element used during the computation. The precision can be specified either directly by setting `Prec` to the desired p-adic precision or by giving a bound B in `Bound`. In the latter case the p-adic precision k will be calculated such that $p^k > B$.

`Stauduhar(G, H, S, B)`

Verbose	`GaloisGroup`	*Maximum :* 5
Verbose	`Invariant`	*Maximum :* 3
`AlwaysTransform`	BoolElt	*Default :* `false`
`Coset`	SeqEnum	*Default :*
`PreCompInvar`	UserProgram	*Default :*

This function gives access to a single step of the Stauduhar method: Let G be a permutation group known to contain the Galois group of the object under investigation with the numbering of the "roots" determined by S. Furthermore, let B be a bound on the absolute value of the complex roots of the object and H be a (maximal) subgroup of G. Under this circumstances, the intrinsic will decide if there is some $g \in G$ such that the Galois group is contained in H^g. The primary return value can be:

1 if the Galois group is proven to be a subgroup of H^g up to precision problems, indicated by the 3rd value

-1 if there is a proof that the Galois group is contained in a proper subgroup of G and maybe in H^g

-2 if the Galois group may be in H^g, but we could not prove that it is in a proper subgroup of G

0 the Galois group is not contained in a conjugate of H.

In case of a non-zero result, the second return value will be the element g conjugating H, the third value will be `true` or `false`, depending on whether the p-adic bound used were proven or heuristic and the fourth value is the invariant used to separate the groups.

The optional parameter `Coset` can be used to pass a transversal of G/H in, while `PreCompInvar` should contain a suitable invariant separating G and H if set.

```
IsInt(x, B, S)
```

Given an element x in the splitting field determined by S and a bound B on the complex absolute value, determine if there exists an integer y such that $y = x$ up the precision of x and such that $|y| < B$. In case such a y exists, it is returned as a second return value.

Example H38E4

A Galois group computation is shown below.

```
> Z:= Integers();
> P<x>:= PolynomialRing(Z);
> G, R, S := GaloisGroup(x^6-108);
> G;
Permutation group G acting on a set of cardinality 6
Order = 6 = 2 * 3
    (1, 5, 3)(2, 6, 4)
    (1, 2)(3, 6)(4, 5)
> R;
[ -58648*$.1 + 53139 + O(11^5), 58648*$.1 - 19478 +
O(11^5), -43755*$.1 - 72617 + O(11^5), 58648*$.1 -
    53139 + O(11^5), -58648*$.1 + 19478 + O(11^5),
    43755*$.1 + 72617 + O(11^5) ]
> S;
GaloisData over Z_11
> time G, _, S := GaloisGroup(x^32-x^16+2);
Time: 65.760
> #G;
2048
```

Some examples for the relative case

```
> load galpols;
> f := PolynomialWithGaloisGroup(9, 14);
> G := GaloisGroup(f);
> TransitiveGroupIdentification(G);
14 9
> M := MaximalOrder(f);
> kM := FieldOfFractions(M);
> f:= Factorisation(Polynomial(kM, f))[2][1];
> f;
$.1^8 + (-2/1*kM.1 + kM.2)*$.1^7 + (-60/1*kM.1 -
    2/1*kM.2 + kM.3)*$.1^6 + (120/1*kM.1 - 60/1*kM.2 -
    2/1*kM.3 + kM.4)*$.1^5 + (980/1*kM.1 + 120/1*kM.2 -
    60/1*kM.3 - 2/1*kM.4 + kM.5)*$.1^4 + (-1808/1*kM.1
    + 980/1*kM.2 + 120/1*kM.3 - 60/1*kM.4 - 2/1*kM.5 +
    kM.6)*$.1^3 + (-4012/1*kM.1 - 1808/1*kM.2 +
    980/1*kM.3 + 120/1*kM.4 - 60/1*kM.5 - 2/1*kM.6 +
    kM.7)*$.1^2 + (4936/1*kM.1 - 4013/1*kM.2 -
    1809/1*kM.3 + 979/1*kM.4 + 118/1*kM.5 - 60/1*kM.6 -
    4/1*kM.7 + 3/1*kM.8)*$.1 - 208769062021/1*kM.1 +
    51146604497/1*kM.2 - 30878218588/1*kM.3 +
    50063809507/1*kM.4 - 52067647419/1*kM.5 -
    94281823910/1*kM.6 + 69906801827/1*kM.7 -
    182364865509/1*kM.8 + 214706745867/1*kM.9
> g, r, p:= GaloisGroup(f);
> TransitiveGroupIdentification(g);
5 8
```

Since g is derived from a factor of the original f, the Galois group should be isomorphic to a subgroup of G:

```
> Subgroups(G:OrderEqual := #g);
Conjugacy classes of subgroups
------------------------------
[1]     Order 8           Length 9
        Permutation group acting on a set of
        cardinality 9
        Order = 8 = 2^3
            (2, 4, 5, 3)(6, 8, 7, 9)
            (2, 7, 5, 6)(3, 9, 4, 8)
            (2, 5)(3, 4)(6, 7)(8, 9)
> IsIsomorphic(g, $1[1]`subgroup);
true Homomorphism of GrpPerm: g, Degree 8, Order 2^3
into GrpPerm: $, Degree 9, Order 2^3 induced by
    (1, 2, 3, 8)(4, 5, 6, 7) |--> (2, 4, 5, 3)(6, 8, 7, 9)
    (1, 7, 3, 5)(2, 6, 8, 4) |--> (2, 7, 5, 6)(3, 9, 4, 8)
```

38.2.1 Straight-line Polynomials

One of the most important tools in the computational Galois theory are invariants, that is multivariate polynomials that are invariant under some permutation group. While invariant theory in general is a rich and classical branch of mathematics, and is supported by a powerful magma module, Chapter 110, the more specific needs in the Galois theory are best met with a different set of functions. Invariants, in this chapter are multivariate polynomials in straight-line representation, the polynomials are represented as programs without branches. The category of this polynomials is of type `RngSLPol` and its elements are of type `RngSLPolElt`. A consequence of this representation is that certain operations are very fast, while others are impossible - or at least very difficult. For example, representing $(a-b)^{1000}(a+b)^{1000} - (a^2-b^2)^{1000}$ is trivial, this is a short program with just a few steps:

1 subtract b from a

2 raise to the 1000th power

3 add a and b

4 raise to the 1000th power

5 multiply the results of steps 2 and 4

6 subtract b^2 from a^2 and raise to the 1000th power

7 subtract the result of step 7 from 5

Now, while it is trivial to evaluate this polynomial at, for example, any pair of elements in any finite ring, it is very difficult to see that, in fact, the polynomial is identical to zero – when expanded as a polynomial.

`SLPolynomialRing(R, n)`

`Global` BoolElt *Default :* `false`

Creates the ring of multivariate straight-line polynomials over the ring R with n indeterminates.

`Name(R, i)`

`R . i`

Return the ith indeterminate of the SL-polynomial ring R.

`BaseRing(R)`

`CoefficientRing(R)`

Return the coefficient ring of R.

`Rank(R)`

Return the rank of the SL-polynomial ring, ie the number of independent indeterminates over the coefficient ring.

`SetEvaluationComparison(R, F, n)`

For a SL-polynomial ring R, prepare "probabilistic" comparison of straight-line polynomials, using evaluation at n tuples drawn at random from the finite field F.

In order to allow a probabilistic test for "equality" of SL-polynomials in places where a strict, deterministic test is not necessary, this allows to compare SL-polynomials through their values at random evaluation points.

`GetEvaluationComparison(R)`

Return the finite field and the number of random samples used to compare polynomials. If `SetEvaluationComparison` has not been called, the 1st return value will be `false` while the second is undefined.

`x * y` `x + y` `x - y` `- x`

`Derivative(x, i)`

The ith partial derivative of the SL-polynomial x.

38.2.2 Invariants

At the core of the computation of Galois groups is the single Stauduhar step where, for a group G and a (maximal) subgroup U the programme decides if the Galois is a subgroup of U - provided it was contained in G. This is achieved by evaluating a G-relative U-invariant polynomial $f \in \mathbf{Z}[x_1, \ldots, x_n]$. In this subsection several functions are collected that allow a user to access MAGMA's internally used invariants. In what follows, an invariant is always a multivariate polynomial f in n indeterminates where n is the degree of G i.e. $G < S_n$. Invariants are represented as straight-line polynomials that allow the very compact representation and fast evaluation of polynomials.

`GaloisGroupInvariant(G, H)`

`DoCost`	BOOLELT	*Default :* `false`
`Worklevel`	RNGINTELT	*Default :* -1
Verbose	`GaloisGroup`	*Maximum :* 3

For subgroups $H < G$ of the symmetric group on n elements, where H is maximal in G and G is transitive, compute a G-relative H-invariant. This is done by carefully comparing certain group theoretical properties of the group pair in question to find invariant polynomials of special types that are easy to evaluate. If this fails, generic invariants will be used.

If `DoCost` is `true`, two values are returned: the first return value in this case is an estimate for the number of multiplications necessary to evaluate the invariant, while the second value is a function that can be evaluated without arguments to compute the invariant. This is done to allow to compare invariants by their computational complexity before actually committing and computing them explicitly as this can be very time consuming.

If `Worklevel` is set to an integer different from -1 only certain types of invariants are tested for suitability for this particular pair of groups. In this case a special return value of `false` indicates that MAGMA was unable to find an invariant at this level. Roughly speaking, the higher the `Worklevel`, the more time-consuming the invariant will be, both in terms of the time spend in finding as well as the time necessary to evaluate the invariant.

`RelativeInvariant(G, H)`

`IsMaximal`	BOOLELT	*Default :* `false`
`Risk`	BOOLELT	*Default :* `false`
Verbose	`Invariant`	*Maximum :* 3

For a pair of subgroups $H < G$ of the symmetric group where H is not necessarily maximal in G, find a G-relative H-invariant polynomial. The computation splits into three phases:

- First, a subgroup chain between H and G is computed such that each step in the chain is a maximal subgroup.

- Second, for each pair $U_i < U_{i+1}$ of maximal subgroups one fixed invariant is computed
- Third, in the last step, the invariants are combined to produce a G-relative H-invariant.

If `IsMaximal` is set to `true`, MAGMA will not compute a subgroup chain but instead assume that H is a maximal subgroup of G. If `Risk` is `true`, then MAGMA will use `GaloisGroupInvariant` to compute invariants on each level. By default, MAGMA will use generic invariants (ie. orbit sums of monomials) on each level. The problem with using special invariants as produced by `GaloisGroupInvariant` is that in the third step we can no longer guarantee that the invariant returned will be a true G-relative H-invariant as the G-stabilizer might be too large. On the other hand, the special invariants are much faster to evaluate and will give the desired result almost always.

`CombineInvariants(G, H1, H2, H3)`

Given a subgroup $G < S_n$ and three maximal subgroups H_1, H_2 and H_3 of G two of which have already known invariants, try to derive an invariant for H_3 from the known ones. The input for H_1 and H_2 consists of a tuple with two (or three) entries, the first specifying the actual subgroupm the second the G-relative H_i-invariant and the optional third a Tschirnhaus transformation that should be done before the invariant is evaluated.

They typical situation in which this function is used is the case of H_1 and H_2 being index 2 subgroups of G. In this case elementary theory immediately guarantees a third subgroup H_3 of index 2. For this function to work, the core of $H_1 \cap H_2$ must be contained in H_3. This is only useful if the index of the core is not too large.

`IsInvariant(F, p)`

`Sign`	BOOLELT	*Default :* `false`

For a multivariate polynomial F in straight-line representation and a permutation p this functions test if $F^p eq F$ with a high probability. In particular, this function will evaluate F at random elements in some large finite field, then permute the evaluation points by p and evaluate again. If the values agree, the polynomial is most likely invariant under p, if they disagree than the polynomial is definitely not invariant. The probability of failure is related to the probability of guessing a zero of a multivariate polynomial at random.

In order to get a proof for the invariants, one can convert F into a standard multivariate polynomial and check directly that this is invariant. However, for the invariants typically constructed in the Galois package, the conversion into a multivariate polynomial will not be possible due to the large degree of the polynomial and the resulting large number of terms.

If `Sign` is set to `true`, the function checks instead for $F^p = -F$.

`Bound(I, B)`

Given a multivariate polynomial I in straight-line representation and an integer B, compute an integer M such that

$$|I(x_1, \ldots, x_n)| \leq M$$

for all complex numbers $|x_i| \leq B$, it it returns a bound for the size of an evaluation of I.

`Bound(I, B)`

Given a multivariate polynomial I in straight-line representation and B, a power series over the integers, compute a power series M such that for all choices of power series x_i such that the coefficients of x_i are bounded in absolute value by those of B we have that the power series

$$I(x_1, \ldots, x_n)$$

has coefficients bounded by those of M.

38.2.3 Subfields and Subfield Towers

The result of a Galois group computation contains, in addition to the Galois group as an abstract group, the explicit action of the group on the roots of the underlying polynomial in some splitting field. This explicit action, together with the availability of invariants for group pairs, can be used to compute arbitrary subfields of the splitting field.

`GaloisSubgroup(K, U)`

`GaloisSubgroup(S, U)`

`GaloisSubgroup(f, U)`

Verbose `Invariant` *Maximum :* 2

Given either a polynomial f or number field K or a successful computation of a Galois group in S and a subgroup $U < G$ where G is the Galois group, find a defining polynomial for the subfield of the splitting field that is fixed by U.

`GaloisQuotient(K, Q)`

`GaloisQuotient(f, Q)`

`GaloisQuotient(S, Q)`

Verbose `Invariant` *Maximum :* 2

Given either a polynomial f or number field K or a successful computation of a Galois group in S and a permutation group Q, find all subfields of the splitting field that have a Galois group isomorphic to Q. This is done by finding all subgroups U of the Galois group G such that the permutation action of G on the cosets G/U is isomorphic to Q.

`GaloisSubfieldTower(S, L)`

`Risk`	BoolElt	*Default :* `false`
`MinBound`	RngIntElt	*Default :* 1
`MaxBound`	RngIntElt	*Default :* ∞
`Inv`	[RngSLPolElt]	*Default :* `false`
Verbose	`GaloisTower`	*Maximum :* 2

For data computed as the third return value of `GaloisGroup` and a subgroup chain $U_1 > U_2 > \ldots > U_s$, compute the corresponding tower of fixed fields K_i that is fixed by the operation of U_i on the roots of f as ordered in S.

Currently, this function only works for polynomials defined over **Q** or absolute extensions of **Q**.

The first return value is the largest number field in the tower, that is the field fixed by the smallest group in the chain as an extension of the fixed field of the second group .... The second return value is a sequence of tuples each containing the data used to generate one step:

- The first item is the invariant used in this step. This corresponds directly to the choice of the primitive element.
- The second item is the Tschirnhaus transformation on this level
- The third item is a transversal of U_i over U_{i+1}, the fixed ordering of which gives the ordering of the "relative conjugates"

The third and fourth return values can be used to algebraically identify arbitrary elements of the splitting field that are defined by multivariate polynomials. The third is a function that takes a vector of p-adic conjugates and returns an algebraic representation of the element, the fourth takes an invariant and computes precision bounds for the precision necessary so that the algebraic recognition will work.

If `Risk` is set to `true`, then for non-maximal subgroup pairs $U_i > U_{i+1}$ the "risky" version of `RelativeInvariant` is used.

The parameter `MinBound` can be used to specify a minimal p-adic precision that should be used internally. This can be used to avoid the calculation of an increase in precision which can be costly. On the other hand, to work in larger precision than necessary also incurs a time penalty.

The parameter `MaxBound` can be used to limit the p-adic precision used internally. Especially when the chain get longer, the internally used precision estimates become more and more pessimistic thus forcing higher and higher precision. In certain cases when it is possible to verify the correctness of the result independently, a smaller precision can speed the computation up considerably.

If the parameter `Inv` is given it should contain a sequence of invariants, the i-th entry need be an U_i relative U_{i+1} invariant. The invariants used correspond almost directly to the relative primitive elements computed at each step in the tower. This is useful in situation where either certain primitive elements are necessary or where certain invariants are known.

`GaloisSplittingField(f)`

Galois	Tup<GrpPerm, [RngElt], GaloisData>	
Roots	BoolElt	*Default :* true
AllAuto	BoolElt	*Default :* false
Stab	BoolElt	*Default :* true
Chain	[GrpPerm]	*Default :* false
Inv	[RngSLPolElt]	*Default :* false
Name	MonStgElt	*Default :* false

For a polynomial f in $\mathbf{Z}[t]$, $\mathbf{Q}[t]$ or over an absolute number field this function computes the splitting field of f as a tower of fields. The various parameter can be used to force certain subfield towers and/ or compute additional data. By default `Stab` is set to `true`, which means that the splitting field will be the tower corresponding to the chain of stabilizers of $\{1\}$, $\{1,2\}$, ..., $\{1,\dots,n\}$. Also by default `Roots` is set to `true`, which means that the roots of f are expressed as elements of the splitting field. If `Roots` is set to `false`, only the field is computed and returned.

The third return value will be the Galois group, the optional fourth value the automorphisms.

If the parameter `Galois` is used, it should contain a list or triplet containing the output of `GaloisGroup(f);`.

If `Chain` is set to a sequence of subgroups, this chain is used to compute a subfield tower. In this case the first elements must be G, the full Galois group. If `Chain` is used, `Inv` can be used to provide the invariants as well.

If `AllAuto` is set to `true`, the full automorphism group of the splitting field is computed as a sequence of sequences giving the all the roots of the relative polynomials.

If `Name` is given, it should be set to a string. In this case the primitive element of the i-subfield in the tower will be called `Name.i`.

Example H38E5

We start with a small example, to illustrate some of the parameters and their influence:

```
> P<x> := PolynomialRing(IntegerRing());
> f := x^3-2;
> GaloisSplittingField(f);
Number Field with defining polynomial $.1^2 +
    $.1*$.1 + $.1^2 over its ground field
[
    $.1,
    $.1,
    -$.1 - $.1
]
Symmetric group acting on a set of cardinality 3
Order = 6 = 2 * 3
```

```
    (1, 2, 3)
    (1, 2)
> K, R, G := $1;
> K:Maximal;
  K
  |
  |
  $1
  |
  |
  Q
K  : $.1^2 + $.1*$.1 + $.1^2
$1 : x^3 - 2
> [x^3 : x in R];
[
    2,
    2,
    2
]
```

The fact that by default all generators are called `$.1` makes this hard to read, so let us assign other names:

```
> GaloisSplittingField(f:Name := "K");
Number Field with defining polynomial K1^2 + K2*K1
    + K2^2 over its ground field
[
    K2,
    K1,
    -K1 - K2
]
Symmetric group G acting on a set of cardinality 3
Order = 6 = 2 * 3
    (1, 2, 3)
    (1, 2)
> (K where K := $1):Maximal;
  $1<K1>
    |
    |
  $2<K2>
    |
    |
    Q
$1 : K1^2 + K2*K1 + K2^2
$2 : x^3 - 2
```

So now we can easily see that the splitting field is a relative quadratic extension of the degree 3 stem field. Now we try a different subgroup chain:

```
> G, r, S := GaloisGroup(f);
> GaloisSplittingField(f:Galois := <G, r, S>,
>     Chain := CompositionSeries(G), Name := "K", AllAuto);
Number Field with defining polynomial K1^3 - 2
over its ground field
[
    K1,
    1/6*K2*K1,
    1/6*(-K2 - 6)*K1
]
Symmetric group G acting on a set of cardinality 3
Order = 6 = 2 * 3
    (1, 2, 3)
    (1, 2)
[
    [
        K2,
        -K2 - 6
    ],
    [
        K1,
        1/6*K2*K1,
        1/6*(-K2 - 6)*K1
    ]
]
> (K where K := $1):Maximal;
  $1<K1>
    |
    |
  $2<K2>
    |
    |
    Q
$1 : K1^3 - 2
$2 : x^2 + 6*x + 36

> f := x^10 - 20*x^8 + 149*x^6 - 519*x^4 + 851*x^2 - 529;
> G, r, S := GaloisGroup(f);G,r,S;
> TransitiveGroupIdentification(G);
8 10
> TransitiveGroupDescription(G);
[2^4]5
```

Thus the Galois group of f is isomorphic to ${}_{1}0T_8$ of type `[2^4]5` and order 80.
We first compute the splitting field directly:

```
>  time _ := SplittingField(f);
```

This takes a long time, mainly because of the type of the Galois group which will require a field tower involving 5 steps and factorisation of a polynomial in such a tower. Now, we try the same by using the Galois information:

```
> time K, R := GaloisSplittingField(f:Name := "K");
Time: 4.740
> K:Maximal;
  K<K1>
    |
    |
  $1<K2>
    |
    |
  $2<K3>
    |
    |
  $3<K4>
    |
    |
    Q
K  : K1^2 + 1/23*(-12*K4^8 + 217*K4^6 - 1374*K4^4
    + 3606*K4^2 - 3381)
$1 : K2^2 + 1/23*(18*K4^8 - 314*K4^6 + 1877*K4^4 -
    4512*K4^2 + 3588)
$2 : K3^2 + 1/23*(-5*K4^8 + 77*K4^6 - 377*K4^4 +
    663*K4^2 - 437)
$3 : x^10 - 20*x^8 + 149*x^6 - 519*x^4 + 851*x^2 -
    529
> [ Evaluate(f, x) eq 0 : x in R];
[ true, true, true, true, true, true, true, true,
true, true ]
```

From the type of the Galois group, `[2^4]5` we expect G to have a normal subgroup A of type C_2^4 such that the quotient G/A is a cyclic group of order 5. To find that subfield we can for example use the Galois computations again:

```
> A := NormalSubgroups(G:OrderEqual := 16)[1]`subgroup;
> GaloisSubgroup(S, A);
x^5 + 1682*x^4 + 715964*x^3 + 99797360*x^2 +
    5206504944*x + 88019915488
(x5 + x10)
```

The second return value $x_5 + x_{10}$ also tells us that the primitive element of the subfield is the sum of two roots of f, namly the 5-th and 10-th in our fixed ordering.
Suppose we want to work in the degree 16 extension over this field, that is we want to work in the fixed field of the trivial subgroup over the field fixed by A:

```
> K, D, Reco, Bnd := GaloisSubfieldTower(S, [A, sub<G|>]);
>  GK := GaloisGroup(K);
>  #GK;
16
>  AbelianInvariant(GK);
```

```
[ 2, 2, 2, 2 ]
```

As an abstract field, K as the splitting field can be described as a quotient of $\mathbf{Q}[x_1, \ldots, x_{10}]/I$ for some suitable ideal I also known as the Galois ideal. On the other hand, by tensoring with some p-adic complection $\mathbf{Q}_p$ we get an embedding of K into $K_p^{\#G} =: \Gamma$. The sequence D that is returned as the 2nd value contains the information necessary to map elements in $\mathbf{Z}[x_1, \ldots, x_{10}]$ via Γ to K. Suppose we want to find x_1, ie. a root of f in K. We first have to get the p-adic image in Γ, with appropriate precision. Step one is to define a suitable multivariate polynomial i that will represent x_1.
The second step is to compute an integer B such that all complex conjugates if i are bounded by B in absolute value. For this we can use information about the size of the roots of f stored in S. The next step now is to get a bound for the p-adic precision.

```
> R := SLPolynomialRing(Integers(), 10);
> i := R.1;
> B := S`max_comp;
> bound := Bnd(B);
```

Now, we need to get the p-adic conjugates of x_1, ie. the image in Γ. The third entry in each of the elements of D contains coset representatives that give the relative conjugates:

```
> rt := [GaloisRoot(i, S:Bound := bound) : i in [1..10]];
> con := CartesianProduct(Reverse([x[3]: x in D]));
> gamma := [Evaluate(i, PermuteSequence(rt, &*p)) : p in con];
> im := Reco(gamma: Bound := B);
> time Evaluate(f, im) eq 0;
```

Before we try to find an automorphism of the base field using this method we want to find the primitive element of the base field of K. The primitive element is essentially given by the 1st part of D. Note that here a Tschirnhaus-tranformation was necessary.

```
> i := D[1][1]; t := D[1][2];
> B := Bound(i, Evaluate(t, S`max_comp));
> bound := Bnd(B);
> rt := [GaloisRoot(i, S:Bound := bound) : i in [1..10]];
> rt := [Evaluate(t, x) : x in rt];
> gamma := [Evaluate(i, PermuteSequence(rt, &*p)) : p in con];
> im := Reco(gamma : Bound := B);
> im;
$.1
> im eq K.2;
true;
```

Now for the automorphism - all we have to change is to permute the roots as we already have the permutation group.

```
> rt := PermuteSequence(rt, Random(G));
> gamma := [Evaluate(i, PermuteSequence(rt, &*p)) : p in con];
> au := Reco(gamma : Bound := B);
> au;
```

```
1/92*(-9*$.1^4 + 386*$.1^3 - 5854*$.1^2 + 37120*$.1 - 82288)
```

In order to "find" arbitrary (integral) elements this way one has to

- define the element as a multivariate polynomial in the roots, i
- with the aid of Bound and the knowledge of the complex roots of f, find a bound B of the complex embeddings of i and use Bnd as above to find a bound M on the p-adic precision
- use the information in D to compute i in Γ, ie all p-adic conjugates in the "correct" ordering
- use Reco to find the algebraic representation.

38.2.4 Solvability by Radicals

For a polynomial $f \in \mathbf{Z}[t]$ with solvable Galois group it is well known that the roots of f can be expressed as nested radicals. On the other hand no good algorithm is known to achieve this. Here we use the explicit action of the Galois group of f as a permutation group on the p-adic roots to compute such an representation.

`SolveByRadicals(f)`

Prime	RngIntElt	*Default :* false
Name	MonStgElt	*Default :* false
Galois	Tup<GrpPerm, [RngElt], GaloisData>	
UseZeta_p	BoolElt	*Default :* false
MaxBound	RngIntElt	*Default :* ∞
Verbose	GaloisTower	*Maximum :* 3

For a polynomial $f \in \mathbf{Z}[t]$ with solvable Galois group, a splitting field as a tower of radical extensions is computed together with algebraic representations of the roots of f as elements in the splitting field. The third return value contains the non-trivial roots of unity which are used.

If the parameter Galois is used, it should contain a list or triplet containing the output of GaloisGroup(f);.

If Prime is used, and Galois is unspecified, the value of Prime is passed onto the Galois group computation and can therefore be used to choose the p-adic field.

If UseZeta_p is set to true, then the expression for the roots of p will contain pure radicals and roots of unity. By default, if UseZeta_p is false, radical expressions for the roots of unit necessary will also be computed.

If MaxBound is given, it will be used as an upper bound for the p-adic precision used internally. Expecially when the radical tower contains many steps, the internally used precision estimates become more and more pessimistic, thus resulting in larger and larger precision.

If Name is set to some string, the i-th level primitive element in the tower will be called Name.i.

```
CyclicToRadical(K, a, z)
```

Let K/k be a number field with cyclic automorphism group of order n generated by $K.1 \to a$ and z be a n-th root of unity in k. This function will return a field L isomorphic to K such that L is a Kummer extension, ie. the defining polynomial for L will be of the form $t^n - b$ for some b in the coefficient field k of K. The second returned value contains the roots of f in L while the third return value contains the roots of unity used.

Example H38E6

```
> P<x> := PolynomialRing(IntegerRing());
> f := x^6 - x^5 - 6*x^4 + 7*x^3 + 4*x^2 - 5*x + 1;
> K, R := SolveByRadicals(f:Name := "K.");
> K:Maximal;
   K<K.1>
     |
     |
  $1<K.2>
     |
     |
  $2<K.3>
     |
     |
  $3<K.4>
     |
     |
     Q
K  : K.1^3 + 1/2*(3*K.4 - 11)*K.2 + 1/2*(-27*K.4 + 23)
$1 : K.2^2 - 5
$2 : K.3^3 - 228*K.4 + 532
$3 : K.4^2 + 3
> [ Evaluate(f, x) eq 0 : x in R];
[ true, true, true, true, true, true ]
```

Note that every step in the tower defining K is radical, ie. given by an equation of type $x^n - a$.

38.2.5 Linear Relations

An important question for various problems is that of finding all linear (additive) relations between the roots of some integral polynomial. While there is a obvious algorithm if the splitting field can be constructed explicitly, there is no obvious way of doing it in general. In this section we provide two algorithms to find those and more general relations and a third that can verify arbitrary relations.

`LinearRelations(f)`

`Proof`	BoolElt	*Default :* `true`
`Galois`	Tup<GrpPerm, [RngElt], GaloisData>	
`UseAction`	BoolElt	*Default :* `false`
`UseLLL`	BoolElt	*Default :* `true`
`Power`	RngIntElt	*Default :* 1
`kMax`	RngIntElt	*Default :* ∞
`LogLambdaMax`	RngIntElt	*Default :* ∞

Given an integral monic polynomial f, this function finds a basis for the module of additive relations between the roots of f in some algebraic closure. The ordering of the roots is the same as chosen by the computation of the Galois group of f, in fact, the roots used are precisely the ones returned by `GaloisRoot`. The output consists of a basis for the relation module encoded in a matrix and the Galois data encoding the ordering of the roots. The algorithm is described in [dGF07].

If `Power` is set to an integer larger than one, the module of relations between the powers of the roots is computed.

`LinearRelations(f, I)`

`Proof`	BoolElt	*Default :* `true`
`Galois`	Tup<GrpPerm, [RngElt], GaloisData>	
`UseAction`	BoolElt	*Default :* `false`
`UseLLL`	BoolElt	*Default :* `true`
`Power`	RngIntElt	*Default :* 1
`kMax`	RngIntElt	*Default :* ∞
`LogLambdaMax`	RngIntElt	*Default :* ∞

Let f be an integral monic polynomial and α_1, ..., α_n be the roots of f in some splitting field in a fixed ordering. The field and the ordering used here are the ones chosen by the computation of the Galois group of f. The splitting field K of f be represented as a quotient $\mathbf{Q}[x_1, \ldots, x_n]/J$ for some suitable ideal J, thus elements in K can be represented as multivariate polynomials in the roots α_i. The sequence I that is passed into this function is interpreted to contain elements in K given via the polynomials in I. This function computes a basis for the module of relations between the elements represented by I. The algorithm is described in [dGF07].

`VerifyRelation(f, F)`

`Galois`	Tup<GrpPerm, [RngElt], GaloisData>	
`kMax`	RngIntElt	*Default :* ∞

Let f be an integral monic polynomial and α_1, ..., α_n be the roots of f in some splitting field in a fixed ordering. The field and the ordering used here are the ones chosen by the computation of the Galois group of f. The splitting field K of f be represented as a quotient $\mathbf{Q}[x_1, \ldots, x_n]/J$ for some suitable ideal J, thus elements in K can be represented as multivariate polynomials in the roots α_i. For a polynomial F in the roots of f, this function verifies if F evaluated at the roots of f equals zero, ie. if F describes a relation between the roots. The algorithm is described in [dGF07].

Example H38E7

The following example originates in a paper [BDE$^+$] where, among other things, polynomials are constructed whose roots have a maximal number of linear dependencies. Note, that the polynomial constructed here is not extremal.

```
> G := ShephardTodd(8);
> R := InvariantRing(G);
> p := PrimaryInvariants(R);
> p;
[
    x1^8 + (-4*i - 4)*x1^7*x2 + 14*i*x1^6*x2^2 + (-14*i +
        14)*x1^5*x2^3 - 21*x1^4*x2^4 + (14*i + 14)*x1^3*x2^5
        - 14*i*x1^2*x2^6 + (4*i - 4)*x1*x2^7 + x2^8,
    x1^12 + (-6*i - 6)*x1^11*x2 + 33*i*x1^10*x2^2 + (-55*i +
        55)*x1^9*x2^3 - 231/2*x1^8*x2^4 + (66*i +
        66)*x1^7*x2^5 + (-66*i + 66)*x1^5*x2^7 -
        231/2*x1^4*x2^8 + (55*i + 55)*x1^3*x2^9 -
        33*i*x1^2*x2^10 + (6*i - 6)*x1*x2^11 + x2^12
]
> f := Resultant(p[1]-2, p[2]-3, 2);
> f := Polynomial(Rationals(), UnivariatePolynomial(f));
> IsPower(f, 4);
true 27/4*x^24 - 135*x^16 + 405*x^12 - 405*x^8 + 162*x^4 - 1
> _, f := $1;
> GG, R, S := GaloisGroup(f);
> #GG;
192
> rel := LinearRelations(f:Galois := <GG, R, S>);
> #rel;
20
> rel[3];
[ 0, 0, 0, -1, 0, -1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
> IR := SLPolynomialRing(Integers(), 24);
> r := &+ [rel[3][i]*IR.i : i in [1..24]];
> r;
(x7 + ((-1 * x4) + (-1 * x6)))
> VerifyRelation(f, r:Galois := <GG, R, S>);
true
```

Now we use this to construct a field of degree 192 over Q where the conjugates of a primitive element span only a 4-dimensional vectorspace. We first define an invariant (a multivariate polynomial) that corresponds to a primitive element. This can be verified by computing all p-adic conjugates and checking that they are pairwise different.

```
> i := &+ [ i*IR.i : i in [1..23]];
> I := [Apply(g, i) : g in GG];
> #{Evaluate(i, R) : i in I};
192
```

So, since $i = \sum i\alpha_i$ is a linear combination of the roots α_i of f, the dimension of the vectorspace spanned by the conjugates of i is 4 again. Note that this is a property of the polynomial not of the field.

```
> i := RelativeInvariant(GG, sub<GG|>);
> I := [Apply(g, i) : g in GG];
> #{Evaluate(i, R) : i in I};
192
>  #LinearRelations(f, I:Galois := <GG, R, S>, Proof := false);
```

38.2.6 Other

`ConjugatesToPowerSums(I)`

For elements in a sequence I, compute the sequence containing the power sums $\sum I_i^j$ for $j = 1, \ldots, \#I$. If I is interpreted to contain the Galois conjugates of some algebraic number (or the roots of some polynomial) then this computed the power sums.

`PowerSumToElementarySymmetric(I)`

Given a sequence I of elements, interpreted as power sums of some algebraic number x, use Newton's relations to compute the elementary symmetric functions in the conjugates of x. In general for this to succeed, the characteristic of the underlying ring needs to be larger than the length of the sequence.

38.3 Subfields

This section contains functions for the computation of all subfields of any number field or all subfields of a given degree of a simple absolute algebraic field or a simple relative extension.

These computations are independent of the computation of the Galois groups, but similiarly there is no limit on the degrees of the field.

The algorithms used are Klüner's method as presented in [Klü95, Klü97, KP97, Klü98] and the newer method of Klüners, van Hoeij and Novocin [vHKN11].

`Subfields(K, n)`

Verbose	`Subfields`	*Maximum :* 4

Given a simple absolute algebraic field or a simple relative extension K and an integer n greater than 1, this function returns a sequence of pairs (2-tuples) containing the subfields of K of degree n together with the embedding homomorphisms of each subfield into K. It is possible that the sequence contains isomorphic fields, but the embeddings will be distinct in such a situation.

`Subfields(K)`

`Al`	MonStgElt	*Default :* "*Default*"
`Current`	BoolElt	*Default :* `false`
`Proof`	RngIntElt	*Default :* 1
Verbose	`Subfields`	*Maximum :* 4

Given an algebraic field K, this function returns a sequence of pairs (2-tuples) containing the subfields of K (except **Q**) together with the embedding homomorphisms of each subfield into K. It is possible that the sequence contains isomorphic fields.

For fields K which are extensions of an algebraic number field the more recent algorithm developed by Klüners and van Hoeij [vHKN11] is used. For fields K which are extensions of **Q**, this algorithm can be selected by setting the parameter `Al` to `"KluenersvanHoeij"` or may be chosen by `"Default"` as the optimal algorithm. By default, the `"Klueners"` algorithm is chosen if the defining polynomial of K factors into large degree factors (with respect to the degree of K) over the residue field of some prime or the coefficients of the defining polynomial are large, otherwise `"KluenersvanHoeij"` is used which is optimal when the defining polynomial only has small degree factors (with respect to the degree of K) over the residue fields of some number of primes, that is, K has a large number of subfields.

38.3.1 The Subfield Lattice

Subfields of number fields can also be retrieved in the form of a lattice from which additional information can be discovered.

`SubfieldLattice(K)`

Verbose	`Subfields`	*Maximum :* 4

The lattice of subfields of an absolute number field K.

`#L`

The number of fields in the lattice L.

`Representative(L)` `Rep(L)`

`Bottom(L)`

The bottom element of the subfield lattice L (this corresponds to **Q**).

`Top(L)`

The top element of the subfield lattice L (this corresponds to the original number field).

`Random(L)`

A random element of the subfield lattice L.

`L ! n`
`L[n]`

The n-th element of the subfield lattice L.

`NumberField(e)`

The number field corresponding to the given subfield lattice element e.

`EmbeddingMap(e)`

The mapping from `NumberField(e)` into the top number field of the subfield lattice.

`Degree(e)`

The (absolute) degree of the number field corresponding to the subfield lattice element e.

`e eq f`

Returns `true` if and only if the subfield lattice elements e and f are equal.

`e subset f`

Returns `true` f and only if e is a subfield of f.

`e * f`

The smallest field containing both e and f.

`e meet f`

The intersection of e and f. This is the largest field common to both of them.

`&meetS`

The intersection of the subfields in the sequence S.

`MaximalSubfields(e)`

The sequence of maximal subfield lattice elements contained in e.

`MinimalOverfields(e)`

The sequence of minimal subfield lattice elements containing e.

Example H38E8

A subfield lattice is shown.

```
> Zx<x> := PolynomialRing(Integers());
> K<a> := NumberField(x^8 - x^4 + 1);
> L := SubfieldLattice(K);
> L;
Subfield Lattice of K
[1] Rational Field
[2] Subfield generated by a root of x^2 - 4*x + 1
[3] Subfield generated by a root of x^2 - 26*x + 241
[4] Subfield generated by a root of x^2 - 10*x + 241
[5] Subfield generated by a root of x^2 + 1
[6] Subfield generated by a root of x^2 - 10*x + 1
[7] Subfield generated by a root of x^2 - 6*x + 1
[8] Subfield generated by a root of x^2 + x + 1
[9] Subfield generated by a root of x^4 + 4*x^2 + 1
[10] Subfield generated by a root of x^4 - x^2 + 1
[11] Subfield generated by a root of x^4 - 8*x^3 + 20*x^2 - 16*x + 1
[12] Subfield generated by a root of x^4 - 4*x^3 + 8*x^2 - 4*x + 1
[13] Subfield generated by a root of x^4 - 6*x^3 + 13*x^2 - 6*x + 1
[14] Subfield generated by a root of x^4 - 4*x^3 + 8*x^2 + 4*x + 1
[15] Subfield generated by a root of x^4 - 2*x^3 + 5*x^2 + 2*x + 1
[16] Subfield generated by a root of x^8 - x^4 + 1
```

Observe that subfields 2, 5 and 8 give the square roots of 2, -1 and 3, respectively. Since these are incommensurate radicals the field generated by them has degree 8 and so must be isomorphic to K.

```
> K2 := AbsoluteField( NumberField([ x^2 + 1, x^2 - 2, x^2 - 3 ]));
> K2;
Number Field with defining polynomial x^8 - 16*x^6 + 88*x^4 + 192*x^2 + 144
over the Rational Field
> IsIsomorphic(K2, K);
true
```

In fact, K is just a "better" version of K2.

```
> OptimizedRepresentation(K2);
Number Field with defining polynomial x^8 - x^4 + 1 over the Rational Field
Mapping from: FldNum: K2 to Number Field with defining polynomial x^8 - x^4 + 1
over the Rational Field
```

38.4 Galois Cohomology

Magma has some rudimentary functions to aid computations in Galois cohomology of number fields.

```
Hilbert90(a, M)
```

`S`	[RngOrdIdl]	*Default :* `false`

Let K be a number field and $M : K \to K$ be an automorphism of K furthermore, denote by k the fixed field of M, thus M generates the automorphism group of the relative cyclic extension K/k. For some element a in K, such that $N_{K/k}(a) = 1$, this function will find some element b such that $a = b/M(b)$. If `S` is given it should contain a sequence of prime ideals such that there exists some b in the S-unit group over S.

```
SUnitCohomologyProcess(S, U)
```

`ClassGroup`	BoolElt	*Default :* `false`
`Ramification`	BoolElt	*Default :* `false`

Let k be a normal number field with (abstract) automorphism group G. For a set of prime ideals S of k, which is closed under the action of the subgroup U of G, a process is created that allows working with the cohomology of the multiplicative group of k - partially represented by a group of S-units. If `ClassGroup` is given, the set S is enlarged to support the current generators of the class group. If `Ramification` is present, then all ramified primes are also included in S.

During the computations with this object the set S can be increased to allow the representation of a larger number of elements.

```
IsGloballySplit(C, l)
```

`Sub`	GrpPerm	*Default :* `false`
Verbose	`Cohomology`	*Maximum :* 2

For a cohomology process C as created by `SUnitCohomologyProcess` and a 2-cocyle $l : U \times U \to k$ given as a Magma-function, decide if l is split, ie. if there exists a 1-cochain $m : U \to k$ such that $\delta m = l$ for the cohomological coboundary map δ. If `Sub` is given it has to be a subgroup of the automorphism group of the number field underlying the cohomology process, otherwise the full automorphism group is used. This allows to restrict a cocyle easily.

As a fixed cocyle l assumes only finitely many values, we can consider it as a cocyle with values in some suitable S-unit group. Similarly, it is exists, m also has values in some S'-unit group for a potentially larger set S'. This function first tries to "remove" ideals from the support of l, to make the set S as small as possible. Then the set is enlarged to make sure that m, if exists, can be found with values in the $S' = S$-unit group. Since the final problem now involves only finitely generated abelian groups, it can be solved by Magma's general cohomology machinery.

```
IsSplitAsIdealAt(I, l)
```

`Sub`	GrpPerm	*Default :* `false`

Let U be a subgroup of the automorphism group G of some number field k, $l : U \times U \to k^*$ a 2-cocyle and I some ideal in k. If `Sub` is given, U is taken to be `Sub`, otherwise $U := G$. Assuming that each element $l(u, v)$ has a valuation at all ideals in the U-orbit of I, ie. we have a unique decomposition of ideals $l(u, v) = J^{x(u,v)}A(u, v)$ for integers $x(u, v)$ and ideals $A(u, v)$ coprime to J for all J in I^U. Then we can use l to define a cocyle with values in I^U which is a finitely generated group. This function determines if this cocycle splits, and if so, computes a 1-cochain with values in I^U for some fixed ordering of I^U. The cochain and I^U are returned on success.

38.5 Bibliography

[**AK99**] Vincenzo Acciaro and Jürgen Klüners. Computing Automorphisms of Abelian Number Fields. *Math. Comp.*, 68(227):1179–1186, 1999.

[**BDE$^+$**] Neil Berry, Arturas Dubickas, Noam Elkies, Bjorn Poonen, and Chris Smyth. The conjugate dimension of algebraic numbers. *Quart. J. Math.*, 55:237–252.

[**dGF07**] Willem de Graaf and Claus Fieker. Finding integral linear dependencies of algebraic numbers and algebraic Lie algebras. *LMS Journal of Computation and Mathematics*, 11, 2007.

[**FK**] C. Fieker and J. Klüners. Computational Galois Theory I: Invariants and Computations over **Q**. in preparation.

[**Gei03**] Katharina Geißler. *Berechnung von Galoisgruppen über Zahl- und Funktionenkörpern.* PhD Thesis, TU-Berlin, 2003. available at URL:http://www.math.tu-berlin.de/˜kant/publications/diss/geissler.pdf.

[**GK00**] Katharina Geißler and Jürgen Klüners. The determination of Galois Groups. *J. Symbolic Comp.*, 30(6):653–674, 2000.

[**Klü95**] Jürgen Klüners. Über die Berechnung von Teilkörpern algebraischer Zahlkörper. Diplomarbeit, Technische Universität Berlin, 1995. URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/klueners.ps.gz.

[**Klü97**] Jürgen Klüners. *Über die Berechnung von Automorphismen und Teilkörpern algebraischer Zahlkörper.* Dissertation, Technische Universität Berlin, 1997. URL:http://www.math.tu-berlin.de/∼kant/publications/diss/diss_jk.ps.gz.

[**Klü98**] Jürgen Klüners. On computing subfields. A detailed description of the algorithm. *J. Theor. Nombres Bordx.*, 2(10):243–271, 1998.

[**KP97**] Jürgen Klüners and Michael E. Pohst. On Computing Subfields. *J. Symbolic Comp.*, 24(3):385–397, 1997.

[**SM85**] Leonhard H. Soicher and John McKay. Computing Galois Groups over the rationals. *J. Number Th.*, 20:273–281, 1985.

[**Sta73**] Richard P. Stauduhar. The determination of Galois Groups. *Math. Comp.*, 27:981–996, 1973.

[**vHKN11**] M. van Hoeij, J. Klüners, and A. Novocin. Generating Subfields. In Anton Leykin, editor, *Proceedings ISSAC 2011*, 2011.

39 CLASS FIELD THEORY

Chapter 39
CLASS FIELD THEORY

39.1 Introduction

This chapter presents the facilities provided in Magma for class field theory. The main objects of interest are abelian extensions of number fields (`FldAb`) and maps between abelian groups and ideal groups.

Class field theory is concerned with the classification of all abelian extensions of a given field. In particular, this covers abelian extensions of number fields, local fields and global function fields. While this chapter deals with the number field case only, Magma can also perform computations in the other cases. For the case of global function fields, see Chapter 43 and for the case of p-adic local fields, Section 47.14.

Abstractly, class field theory parametrizes abelian extensions in terms of abelian groups defined with respect to the base field. In Magma, ray class groups and their quotients are used to define the extensions. Ray class groups and their quotients are always represented as maps between a finite abelian group (`GrpAb`) and the power structure of ideals (`PowIdeal`). The maps are usually obtained as products of the map returned by ray class groups and quotient maps.

In theory, a class field is completely determined by the corresponding class group (map). Currently there is only a small number of invariants that can be computed directly from the map; for most other properties the class field has to be converted into a number field by computing a set of defining equations.

39.1.1 Overview

Class field theory classifies all abelian extensions of a given number field k in terms of quotients of ray class groups. Ray class groups should be thought of as generalized class groups in that they can be defined similarly to class groups:

$$1 \to U \to k^* \to I \to \mathrm{Cl} \to 1$$

where k^* is the multiplicative group of k, U is the group of units, and I the group of fractional ideals. (Recall, the class group is defined as the quotient of the ideals modulo the principal ideals). To define ray class groups we need to refine all of the above terms. Let $o = \mathbf{Z}_k$ be the ring of integers in k and fix an integral ideal $\mathfrak{m}_0$ in o. Furthermore, let $\mathfrak{m}_\infty$ be a subset of the real embeddings of k into $\mathbf{C}$ and, formally, $\mathfrak{m} := (\mathfrak{m}_0, \mathfrak{m}_\infty)$. Then for $x \in k$ define

$$x \bmod^* \mathfrak{m} = 1 \quad \text{iff} \begin{cases} v_{\mathfrak{p}}(x-1) \geq v_{\mathfrak{p}}(\mathfrak{m}_0) & \text{for all prime ideals } \mathfrak{p} \\ s(x) > 0 & \text{for } s \in \mathfrak{m}_\infty \end{cases}$$

Let $I^{\mathfrak{m}}$ denote the group of fractional ideals that are coprime to $\mathfrak{m}_0$, $U_{\mathfrak{m}}$ the units u such that $u \bmod^* \mathfrak{m} = 1$ and $P_{\mathfrak{m}}$ the elements $x \in k$ such that $x \bmod^* \mathfrak{m} = 1$. Then

$$1 \to U_{\mathfrak{m}} \to P_{\mathfrak{m}} \to I^{\mathfrak{m}} \to \mathrm{Cl}_{\mathfrak{m}} \to 1$$

defines the ray class groups modulo $\mathfrak{m}$. For $\mathfrak{m} = (1o, \emptyset)$, this corresponds to the usual class group. Given two moduli $\mathfrak{m}$ and $\mathfrak{n}$ such that $\mathfrak{m}_0 | \mathfrak{n}_0$ and $\mathfrak{m}_\infty \subseteq \mathfrak{n}_\infty$ we have canonical surjections

$$\mathrm{Cl}_{\mathfrak{n}} \to \mathrm{Cl}_{\mathfrak{m}} .$$

Now, let H be a subgroup such that $P_{\mathfrak{m}} < H < I^{\mathfrak{m}}$. There exists a minimal $\mathfrak{n}$ such that the canonical embedding $\mathrm{Cl}_{\mathfrak{m}} / H \to \mathrm{Cl}_{\mathfrak{n}}$ is injective. This $\mathfrak{n}$ is called the conductor of H.

The main theorem of class field theory asserts the following correspondence between abelian extensions K/k of k and quotients of ray class groups: Let K/k be abelian and $G := \mathrm{Gal}(K/k)$ the group of k-automorphisms of K. For each prime ideal $\mathfrak{p}$ of k that is unramified in K, let $\mathrm{Frob}(\mathfrak{p}, K/k)$ be the Frobenius automorphism, i.e. the unique $s \in G$ such that $s(x) = x^N \bmod \mathfrak{p}$ for all $x \in k$. Extend this map multiplicatively to all ideals coprime to $d_{K/k}$, the discriminant of K/k. This is known as the Artin-map and is denoted by $(b, K/k)$. Class field theory asserts the existence of some modulus $\mathfrak{m}$ and a subgroup H such that $G \cong \mathrm{Cl}_{\mathfrak{m}} / H$ by the Artin-map.

Conversely, for each ray class group $\mathrm{Cl}_{\mathfrak{m}}$ and each subgroup $P_{\mathfrak{m}} < H$ there is an abelian extension K/k with the above property.

The correspondence is one-to-one if one restricts to pairs $\mathrm{Cl}_{\mathfrak{m}}$ and H such that $\mathfrak{m}$ is the conductor of H.

For $\mathfrak{m} = (1o, \emptyset)$, the corresponding field K is called the Hilbert class field of k (in the wide sense).

Perhaps best understood is the Hilbert class field. As conjectured by Hilbert and proved by Furtwängler, all ideals of k become principal in the Hilbert class field H_k. Futhermore, H_k is the maximal unramified abelian extension of k.

A different interpretation of H is provided by norm groups. We have

$$H = \langle N_{K/k}(b) | b \in I_K^{\mathfrak{m}} \rangle$$

This result may be used to convert an abelian number field into an abelian extension. In addition, class field theory asserts that, for an arbitrary normal extension K/k, the norm group defined above corresponds to the maximal abelian subfield.

39.1.2 Magma

In Magma maps are used to represent the ideal groups. If necessary, the map can be augmented by supplying $\mathfrak{m}$ i.e. an integral ideal $\mathfrak{m}_0$ and a sequence of indices of the real places in $\mathfrak{m}_\infty$. The map returned as a second return value from `ClassGroup` or `RayClassGroup` carries the necessary information to recover $\mathfrak{m}$, even if this is hidden from the user. As an example, we consider the Hilbert class field of $k := \mathbf{Q}(\alpha)$ with $\alpha^3 + \alpha^2 + 3\alpha - 6 = 0$ with class group C_4.

Example H39E1

```
> k := NumberField(Polynomial([ -6, 3, 1, 1 ]));
```

Now, the easiest way to get the Hilbert class field is to call `HilbertClassField` directly on k:

```
> K := HilbertClassField(k);
> K;
Number Field with defining polynomial $.1^4 + (76*k.1^2 - 420*k.1 - 488)*$.1^2\
 +    32080*k.1^2 + 41984*k.1 + 95168 over k
```

Let us now verify some of the properties, starting with the discriminant. Since K/k is totally unramified, the discriminant of the maximal order should be 1:

```
> O := MaximalOrder(K);
> O;
Maximal Order of Equation Order with defining polynomial x^4 + [-488, -420,
    76]*x^2 + [95168, 41984, 32080] over its ground order
> Discriminant(O);
Ideal
Basis:
[1 0 0]
[0 1 0]
[0 0 1]
```

Now let us check that all ideals of k are principal in K. In order to do so, it is sufficient to demonstrate that a generator of the class group becomes principal:

```
> g, m := ClassGroup(k);
> g;m;
Abelian Group isomorphic to Z/4
Defined on 1 generator
Relations:
    4*g.1 = 0
Mapping from: GrpAb: g to Set of ideals of Maximal Equation Order with
defining polynomial x^3 + x^2 + 3*x - 6 over its ground order
> i := m(g.1);
> I := O!!i;
> IsPrincipal(I);
true
> f,g := IsPrincipal(I);
> g;
[[2193497788678474035456, -1238066307883451022336,
-2319265120953032748288], [394272965034846395136,
-222653406238254306432, -417025104282566694144],
[167087257584, 71708840496, 32825469008], [495632919,
-279332235, -523526213]] / 10917386545536
```

However, to use the more sophisticated functions, the construction of the class fields needs to be done step–by–step. We first create an abelian extension as an object of type `FldAb`. Since we wish to see how much of the class group becomes trivial in the quadratic subfield K_1 of K (capitulates in K_1), we also define this.

```
> aK := AbelianExtension(m);
> g, m := ClassGroup(k);
> q, mq := quo<g | 2*g.1>;
> m2 := Inverse(mq)*m;m2;
Mapping from: GrpAb: q to Set of ideals of Maximal
Equation Order with defining polynomial x^3 + x^2 + 3*x
    - 6 over Z
Composition of Mapping from: GrpAb: q to GrpAb: g and
Mapping from: GrpAb: g to Set of ideals of Maximal
Equation Order with defining polynomial x^3 + x^2 + 3*x
    - 6 over Z
> aK2 := AbelianExtension(m2);
```

This demonstrates a very important technique: the creation of a quotient group of the class group together with the corresponding map.
A few invariants such as degree and discriminant may be calculated directly from `aK` without the (costly) computation of a defining equation.

```
> Discriminant(aK);
Principal Ideal
Generator:
    [1, 0, 0]
[ 4, 4 ]
```

The second return value denotes the signature of the field as an extension of **Q**.
Now we compute a defining equation for `aK2` and see what happens to the class group of k.

```
> O := MaximalOrder(aK2);O;
Maximal Order of Equation Order with defining
polynomial x^2 + [-5, -2, -1] over its ground order
> O!!m(g.1);
> IsPrincipal($1);
false
> IsPrincipal($2^2);
true
```

So only “half” of the class group capitulates in `aK2`, the reminding part collapses in `aK`.

39.2 Creation

The most powerful way to create class fields or abelian extensions in Magma is to use the `AbelianExtension` function that enables the user to create the extension corresponding to some ideal group.

So, before we can describe the creation functions for the class fields, we have to deal with the ideal groups.

39.2.1 Ray Class Groups

The classical approach to class field theory, which is well suited for computation, is based on *ideal groups* which are generalisations of the ideal class group.

In this section we describe in detail how to create *full* ideal groups, mainly ray class groups. As ray class groups are closely related to the unit groups of residue class rings of maximal order, these too are presented here.

In addition to the functions listed here, the `CRT` on page 944 is relevant in this context.

`RayClassGroup(I)`

`RayClassGroup(I, T)`

Given an integral ideal I belonging to the maximal order of a number field, the *ray class group modulo I* is the quotient of the subgroup generated by the ideals coprime to I by the subgroup generated by the principal ideals generated by elements congruent to 1 modulo I and T if present.

The sequence T contains the numbers $[i_1, \ldots, i_r]$ of certain real infinite places. When the sequence is supplied, the generators of the principal ideals must take positive values at the places indicated by T. The sequence T must be strictly ascending containing only positive integers, each not exceeding the number of real embeddings.

This function requires the class group to be known. If it is not already stored, it will be computed in such a way that its correctness is not guaranteed. However, it will almost always be correct. If the user requires a guaranteed result, then the class group must be verified by the user or computed up to the proof level required beforehand.

The ray class group is returned as an abelian group A, together with a mapping between A and a set of representatives for the ray classes.

The algorithm used is a mixture of Pauli's approach following Hasse ([Pau96, HPP97]) and Cohen's method ([CDO96, CDO97, Coh00]).

`RayClassGroup(D)`

Given a divisor (or place) of an absolute number field, compute the Ray class group defined modulo the divisor.

This function requires the class group to be known. If it is not already stored, it will be computed in such a way that its correctness is not guaranteed. However, it will almost always be correct. If the user requires a guaranteed result, then the class group must be verified by the user or computed up to the proof level required beforehand.

The ray class group is returned as an abelian group A, together with a mapping between A and a set of representatives for the ray classes.

The algorithm used is a mixture of Pauli's approach following Hasse ([Pau96, HPP97]) and Cohen's method ([CDO96, CDO97, Coh00]).

Example H39E2

Some ray class groups are computed below. The example merely illustrates the fact that ray class groups tend to grow if their defining modul grows and can be arbitrarily large. This should be compared to class groups where it is rather difficult to give examples of fields having "large" class groups — unless one takes imaginary quadratic fields.

```
> R<x> := PolynomialRing(Integers());
> o := MaximalOrder(x^2-10);
> RayClassGroup(2*o, [1,2]);
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0
Mapping from: Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0 to Set of ideals of o
> RayClassGroup(2*o);
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*$.1 = 0
Mapping from: Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*$.1 = 0 to Set of ideals of o
```

As one can see, the inclusion of the infinite places only added a C_2 factor to the group. In general, a set of infinite places containing n elements can at most add n C_2 factors to the group without infinite places.
Now we enlarge the modulus by small primes. As one can see, the ray class group gets bigger.

```
> RayClassGroup(8*3*5*7*11*13*101*o);
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4 + Z/4 + Z/24 + Z/24 + Z/120 +
Z/600
Defined on 9 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0
    2*$.3 = 0
    8*$.4 = 0
    24*$.5 = 0
    4*$.6 = 0
    120*$.7 = 0
    12*$.8 = 0
    600*$.9 = 0
Mapping from: Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4 + Z/4 + Z/24 +
Z/24 + Z/120 + Z/600
Defined on 9 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0
    2*$.3 = 0
    8*$.4 = 0
    24*$.5 = 0
    4*$.6 = 0
    120*$.7 = 0
    12*$.8 = 0
    600*$.9 = 0 to Set of ideals of o
```

`RayResidueRing(I)`

`RayResidueRing(I, T)`

Given an integral ideal I belonging to the maximal order of a number field, the *ray residue ring modulo* I is the unit group of the maximal order modulo I extended by one C_2 factor for each element of T. The sequence T should be viewed as a condition on the signs of the numbers factored out.

Let I be an integral ideal of an absolute maximal order and let T be a set of real places given by an increasing sequence containing integers i, $1 \leq i \leq r_1$ where r_1 is the number of real zeros of the defining polynomial of the field. This function computes the group of units $\bmod^*(I, T)$. The result is a finite abelian group and a map from the group to the order to which the ideal belongs.

When T is not given the unit group of the residue ring mod m is returned. This is equivalent to formally setting $T := []$ to be the empty sequence.

`RayResidueRing(D)`

Given an effective divisor D of a number field, compute the unit group of the residue class ring defined modulo the divisor, ie. compute the group of elements that for the finite places in the support of D approximate 1 and have positive sign at the real infinite places of the support of D.

```
pSelmerGroup(p, S)
```

Verbose	`ClassGroup`	*Maximum* : 5
`Raw`	BoolElt	*Default* : `false`
`Integral`	BoolElt	*Default* : `true`
`ExtraPrimes`	RngIntElt	*Default* : 0

Let S be a finite set of prime ideals of the absolute maximal order O of some number field k. Let p be a prime number. The p-Selmer group with respect to S is the abelian group defined by:

$$k_p(S) := \{x \in K \mid v_a(x) = 0 \bmod p \ \forall a \notin S\}.$$

Equivalently:

$$k_p(S) := \{x \in K \mid \exists \text{ ideals } a, b : a^p b = (x) \text{ and } b \text{ coprime to } S\}.$$

This function is designed primarily for use in descent calculations. It also plays a role in class field theory: under certain conditions, the maximal abelian extension of k unramified outside S is obtained by adjoining nth roots of the elements of $k_p(S)$.

The optional parameters can be used to fine-tune the algorithm. In order to explain how to use then, we first outline the algorithm. The p-Selmer group is realized as a subgroup of a quotient of a suitable group of $\tilde{S}$-units of the number field, the images of the map returned by `pSelmerGroup` are $\tilde{S}$-units. Initially, $\tilde{S}$ is chosen as S and then enlarged until the p-part of the ideal class group of K is generated by the ideals in S. The parameter `Raw` is related to the same parameter in `SUnitGroup`, see SUnitGroup for more information. If the parameter is set to `true`, the objects returned as images of the `pSelmerGroup` map are exponent vectors that are applied to a fixed sequence of elements to get actual S-units, see the following example for a demonstration.

In addition to changing the return type, `Raw` also implies a reduction of the results, elements returned under `Raw` are reduced by removing the projection of the lattice generated by pth powers of $\tilde{S}$-units. As a side effect of this reduction, elements are no longer guaranteed to be integral. To offset this, the parameter `Integral` can be set to `true`, in which case the sequence of multiplicative generators will be extended to contain uniformizing elements for all ideals in $\tilde{S}$ and the exponent vectors will be supplemented accordingly to achieve integrality.

The last parameter, `ExtraPrimes` can be set to a positive integer to force the set $\tilde{S}$ be larger by at least `ExtraPrimes`. In some examples, the extra primes can be used to find smaller images under the map returned.

Example H39E3

We compute the 3-Selmer group of $\mathbf{Q}(\sqrt{10})$ with respect to the primes above 2, 3, 11:

```
> k := NumberField(Polynomial([-10, 0,1]));
> m := MaximalOrder(k);
> lp := Factorization(2*3*11*m);
> S := [ i[1] : i in lp];
> KpS, mKpS := pSelmerGroup(3, Set(S));
> KpS;
Abelian Group isomorphic to Z/3 + Z/3 + Z/3 + Z/3 + Z/3
Defined on 5 generators
Relations:
    3*KpS.1 = 0
    3*KpS.2 = 0
    3*KpS.3 = 0
    3*KpS.4 = 0
    3*KpS.5 = 0
> mKpS;
Mapping from: RngOrd: m to GrpAb: KpS given by a rule
> mKpS(m!11);
KpS.2
> mKpS(m!11*2);
KpS.2 + 2*KpS.3 + 2*KpS.5
> mKpS(m!11*2*17^3);
KpS.2 + 2*KpS.3 + 2*KpS.5
```

So as long as the argument to `mKpS` is only multiplied by cubes, the image will be stable. Next, we do the same again, but this time using `Raw`:

```
> KpS, mKpS, mB, B := pSelmerGroup(3, Set(S):Raw);
> B;
(-m.1 -11/1*m.1 3/1*m.1 2/1*m.1 13/1*m.1 5/1*m.1 31/1*m.1
    3/1*m.1 - 2/1*m.2 3/1*m.1 + 2/1*m.2 m.1 - 2/1*m.2 m.1
    + 2/1*m.2 m.2 m.1 + m.2 m.1 - m.2 -2/1*m.1 - m.2 m.2
    3/1*m.1 3/1*m.1 11/1*m.1)
> #Eltseq(B);
19
> mB;
Mapping from: GrpAb: KpS to Full RSpace of degree 19 over
Integer Ring given by a rule [no inverse]
> r := KpS.1 + KpS.2 + 2*KpS.4 + KpS.5;
> r @@ mKpS;
396/1*m.1 + 99/1*m.2
> r @ mB;
( 0  1 -2  0  0  0  0  0  0  0  0  0  1  0  1  0  3  0  0)
> PowerProduct(B, $1);
396/1*m.1 + 99/1*m.2
```

39.2.2 Maps

`InducedMap(m1, m2, h, c)`

Given maps $m_1 : G_1 \to I_1$, $m_2 : G_2 \to I_2$ from some finite abelian groups into the ideals of some maximal order, and a map $h : I_1 \to I_2$ on the ideals, compute the map induced by h on the abelian groups.

For this to work, m_1, and m_2 need to be maps that can be used to define abelian extensions. This implies that m_i has to be a composition of maps where the last component is either the map returned by `ClassGroup` or by `RayClassGroup`. The argument c should be a multiple of the minima of the defining moduli.

The result is the map as defined by

```
hom<G_1 -> G_2 | [ h(r_1(G_1.x)) @@ r_2 : x in [1..Ngens(G_1)]]>
```

For larger modules however, this function is much faster than the straightforward approach. This function tries to find a set of "small" generators for both groups. Experience shows that ray class groups (and their quotients) can usually defined by a "small" set of "small" prime ideals. Since solving the discrete logarithm in ray class groups depends upon solving the discrete logarithm for class groups which is quite slow for "large" ideals, it is much faster in general to use this rather roundabout approach. "Small" ideal in this context means "small" norm.

`InducedAutomorphism(r, h, c)`

An abbreviation for `InducedMap(r, r, h, c)`.

Example H39E4

Consider a "large" ray class group over $k := \mathbf{Q}[\sqrt{10}, \zeta_{16}]$

```
> k := NumberField([Polynomial([-10, 0, 1]), CyclotomicPolynomial(16)]);
> k := OptimizedRepresentation(AbsoluteField(k));
> o := MaximalOrder(k);
> ClassGroup(o:Bound := 500);
Abelian Group of order 1
Mapping from: Abelian Group of order 1 to Set of ideals of o
> IndependentUnits(o);
> SetOrderUnitsAreFundamental(o);
> p := &* [113, 193, 241];
> r, mr := RayClassGroup(p*o);
> #r; Ngens(r);
1706131176377019905236218856143547400125963963181004962861678592\
000000000
26
```

The automorphisms of k act on this group. One way of obtaining the action is:

```
> autk := Automorphisms(k);
> time m1 := hom<r -> r | [ autk[2](mr(r.i))@@ mr : i in [1..Ngens(r)]]>;
```

```
Time: 26.300
```

In contrast, using `InducedAutomorphism`:

```
> time InducedAutomorphism(mr, autk[2], p);
Time: 19.080
```

Now we increase p:

```
> p *:= 257*337;
> r, mr := RayClassGroup(p*o);
> #r; Ngens(r);
38317484207550232782125401256286350350388082479558597692663888 51\
82594198717081342286237067274534259909748926334701892829970432 00\
0000000
42
> time m1 := hom<r -> r | [ autk[2](mr(r.i))@@ mr : i in [1..Ngens(r)]]>;
Time: 115.120
> time InducedAutomorphism(mr, autk[2], p);
Time: 82.390
```

For "small" examples the direct approach is much faster, but for large ones, especially if one is only interested in certain quotients, the other approach is faster.

39.2.3 Abelian Extensions

The ultimate goal of class field theory is the classification of all abelian extensions of a given number field. Although the theoretical question was settled in the 1930's, it is still difficult to explicitly compute defining equations for class fields. For extensions of imaginary quadratic fields, there are well known analytic methods available. This section explains the basic operations implemented to create class fields in MAGMA.

```
RayClassField(m)
AbelianExtension(m)
RayClassField(m, I, T)
AbelianExtension(m, I, T)
RayClassField(m, I)
AbelianExtension(m, I)
```

Given a map $m : G \to I_k$ where G is a finite abelian group and I_k is the set of ideals of some absolute maximal order construct the class field defined by m.

More formally, m^{-1} must be a homomorphism from some ray class group R onto an finite abelian group G. If either I or I and T are given, they must define R. This implies that I has to be an integral ideal and that T has to be a sequence containing the relevant infinite places. Otherwise, MAGMA will try to extract this information from m. The class field defined by m has Galois group isomorphic to $R/\ker(m^{-1})$ under the Artin map.

Note that MAGMA cannot check whether the map passed in is valid. If an invalid map is supplied, the output will most likely be garbage.

`AbelianExtension(I)`

Creates the full ray class field modulo the ideal I.

`RayClassField(D)`

Create the full Ray class field defined modulo the divisor D, ie. an abelian extension that is unramified outside the support of D and such that the (abelian) automorphism group is canonically isomorphic to the ray class group modulo D.

`AbelianpExtension(m, p)`

For a map m as in `AbelianExtension` and a prime number p, create the maximal p-field, i.e. the maximal subfield having degree a p–power.

Example H39E5

The abelian extensions of $\mathbf{Q}$ are known to lie in some cyclotomic field.
We demonstrate this by computing the 12-th cyclotomic field using class fields:
Unfortunately, as ray class groups are not defined for $\mathbf{Z}$ in MAGMA, we must work in a degree 1 extension of $\mathbf{Q}$:

```
> x := ext<Rationals()|>.1;
> Q := ext<Rationals()| x-1 :DoLinearExtension>;
> M := MaximalOrder(Q);
```

The ray class group that defines $\mathbf{Q}(\zeta_{12})$ is defined $\mathrm{mod}(12, \infty)$ where ∞ is the unique infinite place of $\mathbf{Q}$. There are at least two ways of looking at this: First, since $Q(\zeta_{12})$ is a totally complex field, the infinite place of $\mathbf{Q}$ must ramify (by convention: $\mathbf{C}$ is ramified over $\mathbf{R}$), so we must include the infinite place in the definition of the ray class group.
Secondly, the ray class group mod(12) without the infinite place is too small. We know $\phi(12) = 4$, $(\mathbf{Z}/12\mathbf{Z})^* = \{1, 5, 7, 11\} = \{\pm 1, \pm 5\}$. As ideals we have $(1) = o = (-1) = (11)$ and $(5) = (-5) = (7)$ so that $\mathrm{Cl}_{12} \cong C_2$. By introducing $T = [1]$, we distinguish ± 1 and ± 5.

```
> G, m := RayClassGroup(12*M, [1]);
> G;
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*G.1 = 0
    2*G.2 = 0
> A := AbelianExtension(m);
> E := EquationOrder(A);
> Ea := SimpleExtension(E);
> Ma := MaximalOrder(Ea);
> Discriminant(Ma);
144
> Factorization(Polynomial(Ma, CyclotomicPolynomial(12)));
[
    <ext<Ma|>.1 + [0, 1, 0, -1], 1>,
    <ext<Ma|>.1 + [0, -1, 0, 0], 1>,
    <ext<Ma|>.1 + [0, 1, 0, 0], 1>,
    <ext<Ma|>.1 + [0, -1, 0, 1], 1>
]
```

The main advantage of this method over the use of the cyclotomic polynomials is the fact that we can directly construct certain subfields:

```
> x := ext<Integers()|>.1;
> M := MaximalOrder(x^2-10);
> G, m := RayClassGroup(3615*M, [1,2]);
> G; m;
Abelian Group isomorphic to Z/2 + Z/2 + Z/4 + Z/80 + Z/240
Defined on 5 generators
Relations:
    4*G.1 = 0
    80*G.2 = 0
    240*G.3 = 0
    2*G.4 = 0
    2*G.5 = 0
Mapping from: GrpAb: G to Set of ideals of M
```

We will only compute the 5-part of this field:

```
> h := hom<G -> G | [5*G.i : i in [1..#Generators(G)]]>;
> Q, mq := quo<G|Image(h)>;
> mm := Inverse(mq) * m;
> mm;
Mapping from: GrpAb: Q to Set of ideals of M
> A := AbelianExtension(mm);
> E := EquationOrder(A);
> E;
Non-simple Equation Order defined by x^5 - [580810, 0]*x^3 +
    [24394020, -40656700]*x^2 + [15187310285, 2799504200]*x
    + [1381891263204, 530506045900], x^5 - [580810, 0]*x^3 +
    [-109192280, 34848600]*x^2 + [30584583385,
    16797025200]*x + [-341203571896, 109180663800] over its
ground order
> C := Components(A);
```

The function `Components` gives a list of cyclic extensions of M that correspond to the cyclic factors of G.

```
> GaloisGroup(NumberField(C[1]));
Permutation group acting on a set of cardinality 5
Order = 5
    (1, 4, 2, 5, 3)
[ -9 + O(41), 15 + O(41), 6 + O(41), 18 + O(41), 11 + O(41) ]
GaloisData over Z_Prime Ideal
Two element generators:
```

```
[41, 0]
[25, 1] - relative case
> GaloisGroup(NumberField(C[2]));
Order = 5
    (1, 4, 2, 5, 3)
[ 38 + O(79), -28 + O(79), -31 + O(79), 27 + O(79), -6 + O(79) ]
GaloisData over Z_Prime Ideal
Two element generators:
[79, 0]
[57, 1] - relative case
```

Thus the Galois group is indeed proven to be $C_5 \times C_5$.

`AbelianExtension(I, P)`

Creates the full ray class field modulo the ideal I and the infinite places in P.

`HilbertClassField(K)`

Creates the Hilbert class field of K, i.e. the maximal unramified abelian extension of K. This is equivalent to `AbelianExtension(1*MaximalOrder(K))`.

`MaximalAbelianSubfield(M)`
`MaximalAbelianSubfield(F)`
`MaximalAbelianSubfield(K)`

`Conductor` [RngOrdIdl, [RngIntElt]] *Default :* [ ]

Let k be the coefficient field of the given number field K. This function creates the maximal abelian extension A of k inside K. If `Conductor` is given, it must contain a multiple of the true conductor. If no value is specified, the discriminant of K is used.

The correctness of this function is based on some heuristics. The algorithm is similar to [Coh00, Algorithm 4.4.3]

`AbelianExtension(K)`
`AbelianExtension(M)`

`Conductor` [RngOrdIdl, [RngIntElt]] *Default :* [ ]

Creates an abelian extension A of k the coefficient field of the input K that is isomorphic to K. If a value for `Conductor` is given, it must contain a multiple of the true conductor, otherwise the discriminant of K is used to be more specific, in case K is a number field, the discriminant of it's maximal order is used as an initial guess, while for field of fractions of orders (K of type `FldOrd`), the defining order gives the initial guess.

In contrast to `MaximalAbelianSubfield`, provided the field is abelian, this function always computes a correct answer.

Example H39E6

We will compute the Hilbert class field of the sextic field defined by a zero of the polynomial $x^6 - 3x^5 + 6x^4 + 93x^3 - 144x^2 - 153x + 2601$ which has a class group isomorphic to $C_3 \times C_3$.

```
> m := LLL(MaximalOrder(Polynomial([2601, -153, -144, 93, 6, -3, 1 ])));
```

The call to LLL is not necessary, but quite frequently class group computations are faster if the order basis is LLL-reduced first.

```
> a,b := ClassGroup(m:Bound := 300);a;
Abelian Group isomorphic to Z/3 + Z/3
Defined on 2 generators
Relations:
    3*a.1 = 0
    3*a.2 = 0
```

`HilbertClassField` on `m` computes a Non-simple number field defining the Hilbert class field of `m`.

```
> H := HilbertClassField(NumberField(m)); H;
Number Field with defining polynomial [ $.1^3 + 1/595*(-460*$.1^5
    + 2026*$.1^4 - 4052*$.1^3 - 52572*$.1^2 + 229338*$.1 -
    529159), $.1^3 + 1/37485*(82*$.1^5 + 4344*$.1^4 + 8805*$.1^3
    + 15990*$.1^2 + 410931*$.1 + 1098693)] over its ground field
> time _ := MaximalOrder(H);
Time: 56.150
```

However, in order to access more of the structural information of `H` we have to create it as an abelian extension, using `b`.

```
> A := AbelianExtension(b);
> HH := NumberField(A);
```

Now we are able to compute the maximal order of `HH` using `A` and verify the discriminant:

```
> time M := MaximalOrder(A:Al := "Discriminant");
Time: 3.260
> Discriminant(M);
Ideal of m
Basis:
[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]
```

Note, that as a side effect, the maximal order of `HH` is now known:

```
> time MaximalOrder(HH);
Maximal Order of Equation Order with defining polynomials x^3 + [841, 841, 312,
    -534, 222, 0], x^3 + [25, -2, -9, 7, -11, -8] over m
```

```
Time: 0.000
```

We will continue this example following the next section.

39.2.4 Binary Operations

The model underlying the class field theory as implemented in Magma is based on the (generalized) ideal class groups. Based on this group-theoretical description, certain binary operations are easily possible:

`A eq B`

Gives two abelian extensions with the same base field, decide if they are the same.

`A subset B`

Gives two abelian extensions with the same base field, decide if they are contained in each other.

`A * B`

Given two abelian extensions with the same base field, find the smallest abelian extension containing both.

`A meet B`

Given two abelian extensions with the same base field, find the largest common subfield.

39.3 Galois Module Structure

If the base field k for class field constructions is normal with respect to some subfield k_0, i.e. k/k_0 is normal with Galois group G and if the defining modulus of the ideal group is G–invariant, then G acts on the ideal group. The following functions view ideal groups as Galois modules. Given an abelian extension A and parameters `All` and `Over`, we will consider this setup:

Let k be the `BaseField` of A and k_1 the coefficient field of k. If `All` is `true`, let $g := \mathrm{Aut}(k/k_1)$, otherwise, $g := \langle \texttt{Over} \rangle$. In both cases we define $k_0 := \mathrm{Fix}(k, g)$. In particular, if k is normal over the coefficient field k_1 then $k_0 = k_1$ and g is the full Galois group.

In general g is not required to contain k_1 automorphisms, so that any subset of the $\mathbf{Q}$ automorphism group is valid as input. By construction, k is normal over k_0, and g acts on the ideals of k. In general however, g does not act on the ideal groups used to define A.

39.3.1 Predicates

`IsAbelian(A)`

`All`	BoolElt	*Default* : `false`
`Over`	[Map]	*Default* : []

Returns `true` if and only if the abelian extension A is abelian over k_0.

`IsNormal(A)`

`All`	BoolElt	*Default* : `false`
`Over`	[Map]	*Default* : []

Returns `true` if and only if the abelian extension A is normal over k_0. This tests whether the defining ideal group is a g-module.

`IsCentral(A)`

`All`	BoolElt	*Default* : `false`
`Over`	[Map]	*Default* : []

Returns `true` if and only if the abelian extension A is central over k_0. If k is cyclic over k_0 then this is equivalent to checking if A is abelian over k_0. This tests whether the defining ideal group is a g–module with trivial action: If N is the norm group of A, the group extension

$$1 \to N \to G \to g \to 1$$

is central.

39.3.2 Constructions

`GenusField(A)`

`All`	BoolElt	*Default* : `false`
`Over`	[Map]	*Default* : []

The genus field is the maximal abelian extension of k_0 that is contained in the abelian extension A. The result of this function is an abelian extension of k_0.

`H2_G_A(A)`

For A such that A is normal over $\mathbf{Q}$ with base field k that is normal too, compute the 2nd cohomology group of the Galois group of k acting on the ideal group defining A.

`NormalSubfields(A)`

`Quot`	SeqEnum[RngIntElt]	*Default* : []

For an abelian extension, normal over Q and defined over a normal number field k as base field, return a list of all normal intermediate fields. If `Quot` is given, restrict to fields where the norm group has the abelian invariants as specified in `Quot`.

`AbelianSubfield(A, U)`

`FixedField(A, U)`

`IsNormal` BoolElt *Default :* `false`

For an abelian extension A with norm group map $G \to I$ for some finite abelian group G and a subgroup $U < G$, define the field corresponding to G/U, ie. the field fixed by U. If `IsNormal` is given then any cohomology information that is present is transferred to the new field - if possible.

`CohomologyModule(A)`

For an abelian extension A defined over some normal field k/Q, compute the cohomology module (see Chapter 68). The maps returned give the transition between the Z-modules used in the cohomology package and the ideal groups used to define A.

The first map returned maps between the automorphism group of k (as an permutation group) and the actual automorphisms of the field. It is obtained as the third return value of `AutomorphismGroup`.

The second map maps between the ideal group used to create A and a standart representation of the same group.

The third map maps between the standart representation of the norm group and the Z-module.

39.4 Conversion to Number Fields

Although in theory an abelian extension "uniquely" defines a number field and therefore all its properties, not all of them are directly accessible (in Magma at least). The functions listed here perform the conversion to a number field, the most important being of course the function that computes defining equations.

`EquationOrder(A)`

Verbose `ClassField` *Maximum :* 5

Given an abelian extension A of a number field, using the algorithm of Fieker ([Fie00, Coh00]) defining equations for A are computed. For each cyclic factor of prime power degree, one polynomial will be constructed. Depending on the size of the cyclic factors encountered, this may be a very lengthy process.

`NumberField(A)`

Converts the abelian extension A into a number field. This is equivalent to `NumberField(EquationOrder(A))`.

`MaximalOrder(A)`

`Al`	MonStgElt	*Default* : "*Kummer*"
`Partial`	BoolElt	*Default* : `false`

Computes the maximal order of the abelian extension A. The result is the same as that given by `MaximalOrder(EquationOrder(A))` but this functions uses the special structure of A and should be much faster in general.

The first step involves computing the maximal orders of each component.

If `Al eq "Kummer"` the maximal order computation uses Kummer theory to compute maximal orders of kummer extensions known to each component then intersects these with the component to gain the maximal order of that component [Sut12].

If `Al eq "Round2"`, the ordinary round 2 maximal order function is used on the components.

If `Al eq "Discriminant"`, the discriminant of the components is passed into the maximal order computation.

In the second step, the components are combined into an approximation of the full maximal order of A.

If `Partial` is `true`, the computations stop at this point, otherwise `MaximalOrder` is again called and the discriminant of A is passed in.

`Components(A)`

Verbose	`ClassField`	*Maximum* : 5

A list of relative extensions is determined. One extension per cyclic factor is computed.

`Generators(A)`

The first return value is a sequence of generating elements for `NumberField(A)`, the second contains the same elements but viewed as elements of the Kummer extension used in the construction. The third list contains the images of the second list under the action of a generator of the automorphism group corresponding to this cyclic factor.

39.5 Invariants

Several invariants of an abelian extension can easily be obtained from the ideal groups without first computing defining equations for the field.

`Discriminant(A)`

Let A be an abelian extension. Based on the conductor-discriminant relation made explicit by [Coh00, Section 3.5.2], the discriminant of the class field A is computed. This does not involve the computation of defining equations. The second return value is the signature of the resulting field.

`AbsoluteDiscriminant(A)`

The absolute discriminant of A as a number field over $\mathbf{Q}$.

`Conductor(A)`

Computes the conductor of the abelian extension A, i.e. the smallest ideal and the smallest set of infinite places that are necessary to define A. The algorithm used is based on [Pau96, HPP97].

`Degree(A)`

The degree of the abelian extension A.

`AbsoluteDegree(A)`

The degree of the abelian extension A over $\mathbf{Q}$.

`CoefficientRing(A)`

`CoefficientField(A)`

`BaseField(A)`

The base field of the abelian extension A, that is `FieldOfFractions(BaseRing(A))`.

`BaseRing(A)`

`CoefficientRing(A)`

The base ring of the abelian extension A, that is the maximal order used to define the underlying ray class group.

`NormGroup(A)`

The norm group (see the definition of norm group on page 998) used to define the abelian extension A.

`DecompositionField(p, A)`

The decomposition field of the finite prime p in the abelian extension A as an abelian (sub)extension.

`DecompositionField(p, A)`

The decomposition field of the place p in the abelian extension A as an abelian extension.

`DecompositionGroup(p, A)`

`DecompositionGroup(p, A)`

The decomposition group of the finite prime p in the abelian extension A. The abelian group returned is a subgroup of the norm group.

```
DecompositionGroup(p, A)
```

The decomposition group of the place p in the abelian extension A. The abelian group returned is a subgroup of the `NormGroup`.

```
DecompositionType(A, p)
```

The "type" of the decomposition of the finite prime ideal p in the abelian extension A as a sequence of pairs $\langle f, e\rangle$ giving the degrees and the ramification indices.

```
DecompositionType(A, p)
```

The "type" of the decomposition of the place p in the abelian extension A as a sequence of pairs $\langle f, e\rangle$ giving the degrees and the ramification indices.

```
DecompositionType(A, p)
```

`Normal` BoolElt *Default :* `false`

The "type" of the decomposition over $\mathbf{Q}$ of the prime number p in the abelian extension A as a sequence of pairs $\langle f, e\rangle$ giving the degrees and the ramification indices. If `Normal` is set to `true` then the algorithm assumes that the base field of A is normal. This is used to speed up the computations.

```
DecompositionTypeFrequency(A, l)
```

`Normal` BoolElt *Default :* `false`

Computes the decomposition type of all elements in l and returns them as a multi-set. The list l must only contain objects for which `DecompositionType` is defined. If `Normal eq true` then the underlying `DecompositionType` function must be able to deal with it too. If `Normal` is set to `true` then the algorithm assumes that the base field of the abelian extension A is normal. This is used to speed up the computations.

```
DecompositionTypeFrequency(A, a, b)
```

`Normal` BoolElt *Default :* `false`

Computes the decomposition type over $\mathbf{Q}$ in the abelian extension A of all prime numbers $a \le p \le b$ and returns them as a multi set.

If `Normal` is set to `true` then the algorithm assumes that the base field of A is normal. This is used to speed up the computations.

39.6 Automorphisms

The group of relative automorphisms of the abelian extension is isomorphic via the Artin map to the ideal group used to define the field. After defining equations are computed, the user can explicitly map ideals that are coprime to the defining modulus to automorphisms of the field.

`ArtinMap(A)`

Returns a map from the defining group (considered as a "subgroup" of the ideals of the base ring) into the automorphisms of the abelian extension A over the base field.

By the defining property of class fields, this map induces an isomorphism on the defining group (as an abelian group) onto the relative automorphisms of A.

Since this function constructs the number field defined by A, this may involve a lengthy calculation.

`FrobeniusAutomorphism(A, p)`

Computes the relative automorphism of the abelian extension A that is the Frobenius automorphism of p. Since this function constructs the number field defined by A, this may involve a lengthy calculation.

`AutomorphismGroup(A)`

`All`	BoolElt	*Default :* `false`
`Over`	[Map]	*Default :* []

If `IsNormal` is true for the abelian extension A with the given parameters, then the automorphism group of A over k_0 is computed. Since this function constructs the number field defined by A, this may involve a lengthy calculation.

`ProbableAutomorphismGroup(A)`

`Factor`	RingIntElt	*Default :* 1

In case of A and it's base field k both begin normal over $\mathbf{Q}$, the automorphism group G of $A/\mathbf{Q}$ is a group extension of the abelian group coming from the definition of A and the automorphism group of $k/\mathbf{Q}$. This functions sets up the corresponding group extension problem and uses `DistinctExtensions` to compute all group theoretical possibilities for G. In case of several possible groups, a further selection based on cycle types and their frequencies is attempted. The optional parameter `Factor` is passed on to `ImproveAutomorphismGroup` to control the amount of time spent on improving the guess.

While this function can be much faster than the direct computation of the automorphism group, the result of this computation is in general not guaranteed. Furthermore, as there are groups that cannot be distinguished by cycle types and their frequencies alone, correctness cannot be achieved by increasing the value of `Factor`. The intended use of this function is to have a (reasonable fast) method of checking is the field under consideration has an interesting group before an unnecessary long call to `AutomorphismGroup` is attempted.

`ImproveAutomorphismGroup(F, E)`

`Factor` RngIntElt *Default :* 1

Given the output of `ProbableAutomorphismGroup` or `ImproveAutomorphismGroup` try to improve the quality of the guess by splitting more primes to get more data for a cycle-type frequency analysis.

Example H39E7

We will demonstrate the use of `ProbableAutomorphismGroup` by investigating extensions of $\mathbf{Q}(\sqrt{10})$:

```
> k := NumberField(Polynomial([-10, 0, 1]));
> R, mR := RayClassGroup(4*3*5*MaximalOrder(k));
> s := [x`subgroup : x in Subgroups(R:Quot := [2,2])];
> a := [ AbelianExtension(Inverse(mq)*mR) where
>                             _, mq := quo<R|x> : x in s ];
> n := [ x : x in a | IsNormal(x:All)];
> ProbableAutomorphismGroup(n[2]);
Finitely presented group on 3 generators
Relations
    $.2^2 = Id($)
    $.3^2 = Id($)
    ($.2, $.3) = Id($)
    ($.1, $.2^-1) = Id($)
    $.1^-1 * $.3 * $.1 * $.3^-1 * $.2^-1 = Id($)
    $.1^2 = Id($)
```

This shows that since there is only one group extension of a V_4 by C_2 with the action induced from the action of the Galois group of k on R, the Automorphism group is already determined. On the other hand, for the first subgroup there are more possibilities:

```
> g, c := ProbableAutomorphismGroup(n[1]);
> #c;
2
```

We will try to find the “correct” guess by looking at the orders of elements which correspond to decomposition types and their frequencies:

```
> {* Order(x) : x in CosetImage(c[1], sub<c[1]|>) *};
{* 1, 2^^7 *}
> {* Order(x) : x in CosetImage(c[2], sub<c[2]|>) *};
{* 1, 2^^3, 4^^4 *}
```

So, if we find a prime of degree 4 we know it’s the second group. Looking at the frequencies, we can be pretty confident that we should be able to find a suitable prime - if it exists. Since among the first 100 primes there is not a single prime with a factor of degree 4 we are pretty confident that the first group is the correct one. By setting the verbose level, we can see how the decision is made:

```
> SetVerbose("ClassField", 2);
> _ := ImproveAutomorphismGroup(n[1], c:Factor := 2);
Orders and multiplicities are  [
    {* 1, 2^^7 *},
    {* 1, 2^^3, 4^^4 *}
]
Probable orders and multiplicities are  {* 1^^3, 2^^34 *}
Error terms are  [ 0.00699329636679632541587354133907,
0.993006703633203674584126458661 ]
```

This indicates that out of 37 primes considered, non had a degree 4 factor, thus we are confident that the first group is the correct one.

`AbsoluteGaloisGroup(A)`

Given an abelian extension, compute it's Galois group over **Q**, ie. the abstract automorphism group of a **Q**-normal closure of A. This function requires the defining equations for A as a number field to be known, but is considerably faster than calling `GaloisGroup` for the number field directly. The group is returned as an permutation group. All roots of the defining polynomial of the coefficient field of A as well as of the defining polynomials of A itself are returned in some local field. The zeros as well as data required for further computations are contained in the 3rd return value.

`TwoCocycle(A)`

For an abelian extension that is normal over Q and defined over a normal base field k/Q, the automorphism group of A/Q is a group extension of the Galois group of k by A. As a group extension it corresponds to an element in the second cohomology group and can be represented by an explicit 2-cocyle with values in the norm group. This function computes such a cocycle. It can be used as an element of the second cohomology group of the cohomology module of A, see `CohomologyModule`.

39.7 Norm Equations

For cyclic fields Hasse's famous norm theorem states that when considering the solvability of norm equations, local solvability everywhere is equivalent to global solvability. Unfortunately, this local-global principle fails in general even for fields with Galois group isomorphic to Klein's group V_4. The extent of this failure is measured by the number knot which is the quotient of the numbers that are everywhere local norms by the global norms. As it turns out, the structure and size of this quotient can be easily computed, so that it is possible to test if Hasse's theorem is sufficient.

As a second consequence, solvability can be decided by looking at the maximal p-subfields for all primes that divide the degree of the field. Even better, a global solution can be obtained by combining solutions from the maximal p-subfields.

It is important to note that local solvability can be decided by analyzing the ideal groups only. Thus, all the "local" functions will avoid computing defining equations and are therefore reasonably fast.

`IsLocalNorm(A, x, p)`

Returns `true` if and only if x is a local norm in the abelian extension A at the finite prime p, i.e. if x is a norm in the extension of the local field obtained by taking the completion at p.

`IsLocalNorm(A, x, i)`

Returns `true` if and only if x is a local norm in the abelian extension A at the infinite prime i.

`IsLocalNorm(A, x, p)`

Returns `true` if and only if x is a local norm in the abelian extension A at the place p, i.e. if x is a norm in the extension of the local fields obtained by taking the completion at p.

`IsLocalNorm(A, x)`

Returns `true` if and only if x is a local norm everywhere in the abelian extension A.

`Knot(A)`

The (number) knot is defined as the quotient group of the group consisting of those elements of the base field of the abelian extension A that are local norms everywhere modulo the elements that are norms. Therefore, if the knot is trivial, an element is a local norm if and only if it is a norm.

Hasse's norm theorem states that for cyclic fields A the knot is always trivial. In general, this is not true for non-cyclic fields.

`NormEquation(A, x)`

Checks if x is a norm, and if so returns an element of the pre-image. As a first step this function verifies if x is a local norm. If x passes this test, the number field of A is computed and by combining solutions of the norm equation in certain subfields a solution in A is constructed. If the knot is non-trivial, the last step may fail.

This function can be extremely time consuming as not only defining equations for A are computed but class groups in some of them. For large A this is much more efficient than just solving the norm equation in the number field.

`IsNorm(A, x)`

As a first step, this function verifies if x is a local norm. If x passes this test, Magma verifies whether the knot is trivial. If it is, `true` is returned. However, if the knot is non-trivial, then the function `NormEquation` is invoked.

Example H39E8

We illustrate the power of the class field theoretic approach with the following example from group theory. We want to solve for elements of norm 2 or 5 in the field $\mathbf{Q}(\zeta_5)(\eta)$ where $\eta^{20} - \zeta_5\eta^{10} + \zeta_5^2 = 0$ over the cyclotomic field. This field has degree 80 over $\mathbf{Q}$ and is therefore far too large for a direct method.

```
> k := CyclotomicField(5);
> kt := ext<k|>;
> K := NumberField(kt.1^20 - k.1*kt.1^10 +k.1^2);
```

Now we convert K into an abelian extension of k:

```
> A := AbelianExtension(K);
> A;
FldAb, defined by (<[590490000000000000000000, 0, 0, 0]>, [])
of structure: Z/2 + Z/10
> Conductor(A);
Ideal
Two element generators:
    [37500, 0, 0, 0]
    [12060, 15120, 7440, 1680]
[]
```

We now recreate A using the smaller conductor. This will significantly speed up the following computations.

```
> m_0, m_inf := $1;
> A := AbelianExtension(K : Conductor := [* m_0, m_inf *]);
```

We first check the local solvability:

```
> IsLocalNorm(A, BaseRing(A)!2);
false
> IsLocalNorm(A, BaseRing(A)!5);
true
> Knot(A);
Abelian Group isomorphic to Z/2
Defined on 1 generator in supergroup:
    $.1 = $.1
Relations:
    2*$.1 = 0
```

Since the knot is isomorphic to a C_2, the local solvability is not sufficient, but we can attempt to solve the equation:

```
> NormEquation(A, BaseRing(A)!5);
true [
    1/10*(-3*zeta_5^3 - 6*zeta_5^2 + zeta_5 -
        7)*$.1*$.2*$.3^4 + 1/10*(9*zeta_5^3 + 3*zeta_5^2 +
        2*zeta_5 + 6)*$.1*$.2*$.3^3 + 1/10*(-6*zeta_5^3 +
        3*zeta_5^2 - 3*zeta_5 + 1)*$.1*$.2*$.3^2 +
        1/10*(-3*zeta_5^3 - 6*zeta_5^2 + zeta_5 -
        7)*$.1*$.2*$.3 + 1/2*(2*zeta_5^3 + zeta_5^2 + zeta_5
        + 2)*$.1*$.2 + 1/10*(-12*zeta_5^3 - 4*zeta_5^2 -
        6*zeta_5 - 13)*$.1*$.3^4 + 1/10*(11*zeta_5^3 +
        2*zeta_5^2 + 13*zeta_5 + 4)*$.1*$.3^3 +
        1/10*(zeta_5^3 + 2*zeta_5^2 - 7*zeta_5 +
        9)*$.1*$.3^2 + 1/10*(-12*zeta_5^3 - 4*zeta_5^2 -
        6*zeta_5 - 13)*$.1*$.3 + 1/2*(2*zeta_5^3 - zeta_5^2
        + 2*zeta_5 + 1)*$.1 + 1/10*(-zeta_5^3 + 3*zeta_5^2 -
        3*zeta_5 + 1)*$.2*$.3^4 + 1/10*(-2*zeta_5^3 +
        zeta_5^2 + 4*zeta_5 - 3)*$.2*$.3^3 +
        1/10*(3*zeta_5^3 - 4*zeta_5^2 - zeta_5 +
        2)*$.2*$.3^2 + 1/10*(-zeta_5^3 + 3*zeta_5^2 -
        3*zeta_5 + 1)*$.2*$.3 + 1/2*(-zeta_5^3 - zeta_5^2 -
        1)*$.2 + 1/10*(-14*zeta_5^3 - 13*zeta_5^2 - 2*zeta_5
        - 21)*$.3^4 + 1/10*(22*zeta_5^3 + 4*zeta_5^2 +
        11*zeta_5 + 13)*$.3^3 + 1/10*(-8*zeta_5^3 +
        9*zeta_5^2 - 9*zeta_5 + 8)*$.3^2 +
        1/10*(-14*zeta_5^3 - 13*zeta_5^2 - 2*zeta_5 -
        21)*$.3 + 1/2*(5*zeta_5^3 + zeta_5^2 + 3*zeta_5 + 4)
]
> _, s := $1;
> Norm(s[1]);
5
```

Thus, the largest field we had to work in was of degree 16.

39.8 Attributes

In this rather technical section, we describe the programming interface to the abelian extension module. Most of the internal representation is available as attributes and may be used to extend the package.

39.8.1 Orders

The only attribute of orders `RngOrd` that is of interest here concerns cyclotomic extensions. The first and usually most time consuming step while computing defining extensions is to adjoin certain roots of unity and to compute class groups of these rather large fields.

The cyclotomic extensions are stored and are available via an attribute. This attribute is a read-only attribute.

`o‘CyclotomicExtensions`

If defined, `CyclotomicExtensions` is a list of records, each containing data for one cyclotomic extension, i.e. for an order o this is an extension of the form $O := o[\zeta_l]$ for a prime power $l = p^n$. The components are

`Abs`	: a maximal order Oa of O given as an absolute extension
`Rel`	: O as an extension of o
`p2n`	: the "l", i.e. the cyclotomic order
`Zeta`	: a primitive lth root of unity as an element of the absolute extension `Abs`
`Aut`	: a list of records describing generators for the automorphism group of O over o. If p is an odd prime then the list will always be of length 1. This list consists of the following components:
`Aut‘Abs`	: the automorphism as an automorphism of Oa
`Aut‘Rel`	: the automorphism as an automorphism of O. Note that in general this is not an o-automorphism, as o itself may contain roots of unity.
`Aut‘Order`	: the order of the automorphism
`Aut‘r`	: the image of ζ_l, this automorphism sends ζ_l to ζ_l^r.

Due to possible common subfields of o and $\mathbf{Q}(\zeta_k)$, the degree of O over o may be smaller than expected. Furthermore, Oa will be in optimized representation.

Example H39E9

We will demonstrate this with the Hilbert class field of $\mathbf{Q}(\sqrt{-1001})$:

```
> Zx<x> := PolynomialRing(Integers());
> k := NumberField(x^2+1001);
> g, m :=ClassGroup(k); g;
Abelian Group isomorphic to Z/2 + Z/2 + Z/10
Defined on 3 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0
    10*$.3 = 0
> K := HilbertClassField(k);
Number Field with defining polynomial [ $.1^2 - 2*k.1 + 136,
    $.1^2 - 4*k.1 - 47, $.1^2 + 2*k.1 - 136, $.1^5 + 37210*$.1^3
    + (-2104500*k.1 + 61148840)*$.1^2 + (-292068000*k.1 +
    14636593760)*$.1 + 305212632000*k.1 - 1779009360128] over k
> o := MaximalOrder(k);
> c := o‘CyclotomicExtensions;
> #c;
2
```

Since there are two non-isomorphic direct cyclic factors of prime power order in the class group of k, we have two cyclotomic extensions stored in o. One has order 2 and the other has order 5:

```
> c[1]‘p2n;
```

```
2
> c[2]`p2n;
5
```

Since the order 2 extension is essentially trivial, we will discuss the other extension in detail.

```
> c[2]`Abs;
Maximal Order of Equation Order with defining polynomial x^8 +
    4*x^7 + 4016*x^6 + 12034*x^5 + 6032056*x^4 + 12044060*x^3 +
    4016040045*x^2 + 4010020020*x + 1000000005005 over Z
> c[2]`Rel;
Maximal Equation Order with defining polynomial x^4 + x^3 + x^2 +
    x + [1, 0] over o
```

The relative extension is generated by a root of the 5th-cyclotomic polynomial of degree 4, so there are no common subfields. The corresponding absolute extension has degree 8. As one can see, the class group (at least a conditional class group) is known:

```
> c[2]`Abs:Maximal;
   F[1]
     |
   F[2]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     x^8 + 4*x^7 + 4016*x^6 + 12034*x^5 + 6032056*x^4 +
    12044060*x^3 + 4016040045*x^2 + 4010020020*x + 1000000005005
Discriminant: 4016024016004000000
Index: 41000908716764795359 81/1
Class Number 12800
Class Group Structure C2 * C4 * C20 * C80
Signature: [0, 4]
```

We will illustrate the `Aut` entries by showing their effect on ζ_5. Since the maps operate between the number fields rather than the orders, we will coerce the results back into the orders for clarity. The only difference between `Rel` and `Abs` is that `Rel` implements the automorphism in the relative extension.

```
> Oa := c[2]`Abs;
> z := c[2]`Zeta; z;
[0, 1, 0, 0, 0, 0, 0, 0]
> Oa!c[2]`Aut[1]`Abs(z);
[0, 0, 1, 0, 0, 0, 0, 0]
> z^c[2]`Aut[1]`r;
[0, 0, 1, 0, 0, 0, 0, 0]
```

The `Order` entry gives the order of the automorphism:

```
> c[2]`Aut[1]`Order;
4
```

```
> Oa!c[2]`Aut[1]`Abs(z);
[0, 0, 1, 0, 0, 0, 0, 0]
> Oa!(c[2]`Aut[1]`Abs($1));
[-1, -1, -1, -1, 0, 0, 0, 0]
> Oa!(c[2]`Aut[1]`Abs($1));
[0, 0, 0, 1, 0, 0, 0, 0]
> Oa!(c[2]`Aut[1]`Abs($1));
[0, 1, 0, 0, 0, 0, 0, 0]
```

As a by–product one can see the optimized representation: the first four basis elements are clearly 1, ζ_5, ζ_5^2, ζ_5^3 which are the T_2 shortest integral elements in m.

39.8.2 Abelian Extensions

Abelian extensions have several attributes. Most of them are only useful in programming.

`A`Components`

This read-only atribute contains a record for each cyclic factor of prime power degree that occurs in the abelian extension. In order to describe the contents we have to fix some definitions. Let o be the base ring of the abelian extension A. Then O will be the maximal order of the cyclotomic extension $o[\zeta_l]$ as an extensions of $\mathbf{Z}$. The algorithm for the computation of defining extensions will firstly compute a generator $a \in O$ such that $O(a^{1/l})$ equals $A(\zeta_l)$. This element will be a certain S-unit of O.

The components are

`Basis` : a matrix B containing order elements. This represents a "multiplicative basis" for the generator a and all the S-units that are used.

`GenRaw` : the exponent vector G defining a, i.e.

$$a = \prod_i {B_{1,i}}^{G_{i,1}}.$$

`UnitsRaw` : a matrix U defining a basis for the group of S-units, i.e.

$$u_j = \prod_i {B_{1,i}}^{U_{j,i}}.$$

`S` : a list containing the prime ideals of S

`Gen` : a as an element of Oa

`GenAut` : the image of a generator for the class field (an image of `Class Field.2`) in the field of fractions of O under a generator for the cyclic group corresponding to this component.

`GenInv` : $1/a$ as an element of the field of fractions of Oa

`O` : the big Kummer extension $Oa(a^{1/l})$

`ClassField`	: an equation order for the cyclic extension corresponding to this component. This will be an extension of o.
`Artin`	: the Artin map on the big Kummer extension $o[\zeta_l](U_S^{1/l})$ where the automorphisms are represented as elements in some abelian group of type $C_l^{\#S}$.

A`DefiningGroup

A`NormGroup

These two attributes give access to the ideal group used to define A, and they have the same structure. `DefiningGroup` is the group used to create A in the first place whereas `NormGroup` contains the group defined modulo the conductor. The user needs to call the function `Conductor` in order to define this component. These attributes are read-only.

The records contain the following components:

`Map`	: the map as passed into the `AbelianExtension` constructor, respectively, an equivalent map.
`m0`	: the "finite" part of the modulus underlying `Map`
`m_inf`	: the "infinite" part of the modulus underlying `Map`
`RcgMap`	: if present, contains the map returned from the call to `RayClassGroup` with `m0` and `m_inf`.
`GrpMap`	: if present, the "rest" of `Map`, i.e. `Map` = `RcgMap` $\circ$ `GrpMap`.

A`IsAbelian

Stores the result of a call to `IsAbelian` with parameter `All := true`.

A`IsNormal

Stores the result of a call to `IsNormal` with parameter `All := true`.

A`IsCentral

Stores the result of a call to `IsCentral` with parameter `All := true`.

Example H39E10

To illustrate the preceeding examples we will investigate the 3-part of the ray class field modulo 36 over $\mathbf{Z}[\sqrt{10}]$:

```
> Zx<x> := PolynomialRing(Integers());
> o := MaximalOrder(x^2+10);
> r, mr := RayClassGroup(36*o);
> q, mq := quo<r| [r.i*3 : i in [1..Ngens(r)]]>;
> A := AbelianExtension(Inverse(mq)*mr); A;
FldAb, defined by (<[36, 0]>, [])
```

```
of structure: Z/3 + Z/3
```

At this stage, the only attribute that is assigned is `DefiningGroup`.

```
> la := GetAttributes(FldAb);
> [ <assigned A``i, i> : i in la];
[ <false, Components>, <true, DefiningGroup>, <false, IsAbelian>,
<false, IsCentral>, <false, IsNormal>, <false, NormGroup> ]
```

So let us have a closer look at `DefiningGroup`:

```
> d := A`DefiningGroup;
> d;
rec<recformat<Map, m0, m_inf, RcgMap, GrpMap> | Map := Mapping
from: GrpAb: q to Set of ideals of o
Composition of Mapping from: GrpAb: q to GrpAb: r and
Mapping from: GrpAb: r to Set of ideals of o, m0 := Principal
Ideal of o
Generator:
    [36, 0], m_inf := [], RcgMap := Mapping from: GrpAb: r to Set
of ideals of o, GrpMap := Mapping from: GrpAb: q to GrpAb: r>
```

As one can see, `Map` is the original map used to define A. Note that this is the composition of `RcgMap` and `GrpMap`, the first being equivalent to `mr`, the second to `mq`. Furthermore, the defining modulus is now available in `m0` and `m_inf`.
After having set up A, we now need to transform it into a number field in order for the `Components` to be assigned:

```
> K := NumberField(A);
> K;
Number Field with defining polynomial [ $.1^3 - 3*$.1 - 6*$.1 -
    11, $.1^3 - 3*$.1 - 6*$.1 + 11] over its ground field
> c := A`Components;
> #c;
2
```

We will focus on the first component only.

```
> B := c[1]`Basis;
> a := &* [ B[1][i] ^ c[1]`GenRaw[i][1] : i in [1..Ncols(B)]];
-11/1*$.2 + 4/1*$.3 + 2/1*$.4
> c[1]`Gen;
[0, -11, 4, 2]
> $1 eq $2;
true
> c[2]`GenInv * a eq 1;
true
```

In S we need all primes dividing the degree 9 of our extension, all primes dividing the modulus 36 and enough primes to generate the 3-part of the class group. Since the class group can be generated by the prime ideal over 2, this leaves us with only two prime ideals in S. Since k is imaginary quadratic, the base field $k[\zeta_3]$ is totally complex of degree 4 which implies unit rank 1.

So the free rank of our S-unit group will be 3, and since we have to take care of the torsion unit, `GenRaw` will have 4 columns.

```
> #c[1]`S;
2
> UnitRank(o);
1
> Ncols(c[1]`UnitsRaw);
4
> U := c[1]`UnitsRaw;
> u := [ &* [B[1][i] ^ U[i][j] : i in [1..Ncols(B)]] : j in [1..4]];
```

Now u_1 should be a fundamental unit for Oa, u_2 and u_3 are S-units and u_4 is the torsion unit, since we adjoin the 3rd-roots of unity, this will be a 6th root of unity.

```
> Oa := Parent(B[1][1]);
> Oa!u[1];
[0, -11, 4, 2]
> IsUnit($1);
true
> Decomposition(u[2]);
[
    <Prime Ideal of Oa
    Two element generators:
        [3, 0, 0, 0]
        [-9, 0, 1, 2], 1>
]
> u[4]^6 eq 1;
```

Now O should be an extension of Oa:

```
> c[1]`O:Maximal;
         F[1]
           /
          /
      E2[1]
        /
       /
   E1[1]
  /
 /
Q
F  [ 1]     x^3 + [[0, -4], [11, -2]]
E 2[ 1]     x^2 + x + [1, 0]
E 1[ 1]     x^2 + 10
Index : <[[1, 0], [0, 0]]>
```

The class field K we are seeking is a subfield of O:

```
> c[1]`ClassField;
Equation Order with defining polynomial x^3 + [-3, 0]*x + [-11,
    -6] over o
> g := c[1]`ClassField.2;
> c[1]`O!g;
[[[0, 0], [0, 0]], [[1, 0], [0, 0]], [[11, 2], [11, -2]]]
```

Note that the absolute term of the defining polynomial is a. Since `GenAut` is an image of g, its minimal polynomial should be the same as that of g:

```
> MinimalPolynomial(g);
$.1^3 - 3/1*o.1*$.1 - 11/1*o.1 - 6/1*o.2
> Evaluate($1, c[1]`GenAut);
0
```

39.9 Group Theoretic Functions

39.9.1 Generic Groups

Quite frequently in computational algebra one constructs a set of objects that generate a group under some operation. Generic groups are finite groups that are defined by generators that have implicit relations. In order to use them, one has to provide a function for the multiplication of two elements and one to check equality. If known, the identity object can also be passed in.

Generic groups are used in the class field package for the automorphism groups. A frequent situation is that one knows certain automorphisms (as maps) and would like to get the group generated by them. If the group is reasonably small, this can be done using the functions in this section.

All functions here rely on the group being small enough to allow complete enumeration of all elements.

The main application are situations where multiplication of the actual objects is time consuming so one would like to transfer as much as possible to some abstract finite group.

`GenericGroup(X)`

Mult	Intrinsic	*Default :* '*'
Eq	Intrinsic	*Default :* 'eq'
Id	Any	*Default :*
Verbose	GrpGen	*Maximum :* 3

Creates the group G generated by the elements of `X`. The function assumes that the group is finite. The second return value is a map from G onto a list of elements of G which are of the same type as the elements of `X`.

Since this function will enumerate all group elements, the group cannot be too large.

```
AddGenerator(G, x)
```

Verbose `GrpGen` *Maximum* : 3

Adds a new generator `x` to G. If `x` was already in G, the value `false` is returned and the other return values are unassigned. Otherwise, the new group and the corresponding map is returned.

G has to be a generic group as returned by `GenericGroup`.

The function applies a version of Dimino's algorithm [But91a] to find all elements of G with as few operations as possible.

```
FindGenerators(G)
```

Given a generic group G as returned by `GenericGroup`, find a small set of generators.

39.10 Bibliography

[**But91a**] Gregory Butler. *Dimino's Algorithm*, pages 13 – 23. Volume 559 of *LNCS* [But91b], 1991.

[**But91b**] Gregory Butler. *Fundamental Algorithms for Permutation Groups*, volume 559 of *LNCS*. Springer-Verlag, 1991.

[**CDO96**] Henri Cohen, Francisco Diaz y Diaz, and Michel Olivier. Computing Ray Class Groups, Conductors and Discriminants. In Cohen [Coh96], pages 52–59.

[**CDO97**] Henri Cohen, Francisco Diaz y Diaz, and Michel Olivier. Computing Ray Class Groups, Conductors and Discriminants. *Submitted to Math. Comp.*, 1997.

[**Coh96**] Henri Cohen, editor. *ANTS II*, volume 1122 of *LNCS*. Springer-Verlag, 1996.

[**Coh00**] Henri Cohen. *Advanced Topics in Computational Number Theory.* Springer, Berlin–Heidelberg–New York, 2000.

[**Fie00**] Claus Fieker. Computing Class Fields via the Artin Map. *Math. Comput.*, 70(235):1293–1303, 2000.

[**HPP97**] Florian Heß, Sebastian Pauli, and Michael E. Pohst. On the computation of the multiplicative group of residue class rings. *Math. Comp.*, 1997.

[**Pau96**] Sebastian Pauli. Zur Berechnung von Strahlklassengruppen. Diplomarbeit, Technische Universität Berlin, 1996.
URL:http://www.math.tu-berlin.de/~kant/publications/diplom/pauli.ps.gz.

[**Sut12**] Nicole Sutherland. Efficient Computation of Maximal Orders of Radical (including Kummer) Extensions. *Journal of Symbolic Computation*, 47:552–567, 2012.

40 ALGEBRAICALLY CLOSED FIELDS

Chapter 40
ALGEBRAICALLY CLOSED FIELDS

40.1 Introduction

Magma contains a system [Ste02, Ste10] for computing with algebraically closed fields, which have the property that they always contain all the roots of any polynomial defined over them. It is of course not possible to construct explicitly the closure of a field, but the system works by automatically constructing larger and larger algebraic extensions of an original base field as needed during a computation, thus giving the illusion of computing in the algebraic closure of the base field.

Such a system was already proposed before (the D5 system [DDD85]), but this has difficulty with the parallelism which occurs when one must compute with several conjugates of a root of a reducible polynomial, leading to situations where a certain expression evaluated at a root is invertible but evaluated at a conjugate of that root is not invertible.

Magma's system has no such problem and one can compute with the field just like any other field in Magma; all standard algorithms which work over generic fields or which use factorization work automatically without having to be adapted to handle the many conjugates of a root.

Especially significant is also the fact that all the Gröbner basis algorithms work well over such fields. One can compute the variety of any zero-dimensional multivariate polynomial ideal over the algebraic closure of its base field. Puiseux expansions of polynomials are also successfully computed using an algebraically closed field.

40.2 Representation

An algebraically closed field is based on an affine algebra (or quotient ring of a multivariate polynomial ring by an ideal of "relation" polynomials). The defining polynomials of this affine algebra are not necessarily irreducible – the system avoids factorization over an algebraic number field when possible, and automatically splits the defining polynomials of the affine algebra when factors are found during computations with the field. These factors often arise automatically because of the structure of the algorithm which is computing over the field.

Many technical optimizations have been designed to make the system practical. For example, the most expensive arithmetic operation by far in the whole system is, quite surprisingly, the testing of whether an element of the field is zero or not (which is utterly trivial for most other rings, of course)! To allow for the parallelism amongst conjugates to work, Magma performs a recursive GCD computation with the element, considered as a polynomial in its highest variable, and the appropriate defining polynomial, to determine the result. If the GCD is non-trivial, then this forces a splitting of the defining polynomial, all elements of the field are reduced, and the original element may now be zero. More details concerning the internal design of the system can be found in [Ste10].

Care must be taken with the interpretation of the roots of a polynomial in this system. The roots of polynomials are only defined algebraically, and the user may wish to identify them with some particular elements of the complex field, for example, but one cannot assume that the system will follow the embedding one wishes. An example will demonstrate. Suppose that α is a root of $x^2 - 2$, β is a root of $x^2 - 3$, and γ is a root of $x^2 - 6$. Does $\gamma = \alpha \cdot \beta$ or does $\gamma = -\alpha \cdot \beta$? The system will have to make a choice between the two possibilities if the situation arises, but the choice which it will make cannot be predicted beforehand. That is, even if we might like to interpret things as $\alpha = \sqrt{2}$, $\beta = \sqrt{3}$ and $\gamma = \sqrt{6}$ (referring to the positive real roots in each case), it cannot be assumed that this will hold in the particular algebraically closed field, since the roots are only defined algebraically.

In the following descriptions, the adjective *invariable*, applied to a value, means that the mathematical value of a function will not change, despite simplifications, etc. which may occur after the function is called. That is, the value returned by the function may be considered constant with respect to the `eq` operator. A value may print differently later (because of simplifications of the field), but the mathematical value will never change.

In contrast, the adjective *variable* means that the mathematical value of a function may change (as a result of simplifications). This is usually only a property of access-like functions like `Degree(A, v)` (see below). But the fact that most functions below have invariable return values enables the illusion of a true field to be sustained.

Separate algebraic fields should be created for separate problems which involve root taking (if the roots of the different problems are unrelated). Otherwise it may be unnecessarily expensive to compute in the field with roots of unrelated polynomials.

Currently, the base field of an algebraic closure may be a finite field, the rational field **Q**, or a rational function field over a finite field or **Q**.

40.3 Creation of Structures

`AlgebraicClosure(K)`

Create the algebraic closure **A** of the field K. Currently K may be **Q**, a finite field or a rational function field over a finite field or the rational field.

`AlgebraicClosure()`

Create the algebraic closure **A** of the rational field **Q**.

`AssignNamePrefix(A, S)`

(Procedure.) Given an algebraically closed field A and a string S, reassign the string prefix of the names of A to be S. By default, this prefix is `"r"`. When new variables are introduced by root taking, they are named by this prefix with the number appended (see `A.i` below).

40.4 Creation of Elements

The usual way of creating elements within an algebraically closed field A is by coercion from the base field into A, or by construction of roots of polynomials over A (and this may be done indirectly via other functions).

40.4.1 Coercion

`A ! a`

Given a finite field A create the element specified by a; here a is allowed to be an element coercible into A, which means that a may be

(i) an element of A;

(ii) an integer or rational.

`One(A)` `Identity(A)`

`Zero(A)` `Representative(A)`

These generic functions create `A!1`, `A!1`, `A!0` and `A!0`, respectively.

40.4.2 Roots

`Roots(f)`

`Roots(f, A)`

`Max` RngIntElt *Default :*

Given a polynomial f over an algebraically closed field A, or given a polynomial f over some subring of A together with A itself, this function computes all roots of f in A, and returns a sorted sequence of tuples (pairs), each consisting of a root of f in A and its multiplicity. Since A is algebraically closed, f always splits completely.

If the parameter `Max` is set to a non-negative number m, at most m roots are returned. This feature can be quite useful when one wishes, say, only one root of a polynomial and not all the conjugates of the root, as they will cause the field to have more variables than necessary and this can make the full simplification of the field much more difficult later (if that is requested).

Note that the function `Factorization(f)` is also supported, and simply returns the linear factors and multiplicities corresponding to the roots returned by `Roots(f)`.

`RootOfUnity(n, A)`

Return a primitive n-th root of unity in A, i.e., an element $\omega \in A$ such that $\omega^n = 1$ and $\omega^i \neq 1$ for $1 \leq i < n$. This always exists since the field is algebraically closed, and the return value is invariable. This function is equivalent to `Roots(CyclotomicPolynomial(n), A: Max := 1)[1, 1]`.

`SquareRoot(a)`

`Sqrt(a)`

A square root of the element a from the field A, i.e., an element y of A such that $y^2 = a$. A square root always exists since the field is algebraically closed, and the return value is invariable.

`IsSquare(a)`

Return `true` and a square root of the element a from the field A, i.e., `true` and an element y of A such that $y^2 = a$. A square root always exists since the field is algebraically closed, and the return value is invariable.

`Root(a, n)`

Return an n-th root of the element a from the field A, i.e., an element y of A such that $y^n = a$. A root always exists since the field is algebraically closed, and the return value is invariable.

`IsPower(a, n)`

Return `true` and an n-th root of the element a from the field A, i.e., `true` and an element y of A such that $y^n = a$. A root always exists since the field is algebraically closed, and the return value is invariable.

40.4.3 Variables

`A . i`

Return the i-th variable of A. i must be between 1 and the rank of A (the current number of variables in A). Initially A has no variables, and new variables are only created by calling `Roots` above (or similar functions such as `Sqrt`). As long as `Prune` or `Absolutize` are not called (which shift the variable numbers – see below), the return value of this function is invariable, so $A.i$ for fixed i will always return the same mathematical object despite any simplifications or constructions of new roots. New roots are always assigned higher generator numbers.

Example H40E1

We first show the most common way of creating roots: by the `Roots` function.

```
> A := AlgebraicClosure();
> P<x> := PolynomialRing(IntegerRing());
> r := Roots(x^3 + x + 1, A);
> r;
[
    <r1, 1>,
    <r2, 1>,
    <r3, 1>
]
> A;
```

```
Algebraically closed field with 3 variables
Defining relations:
[
    r3^3 + r3 + 1,
    r2^3 + r2 + 1,
    r1^3 + r1 + 1
]
> a := r[1,1];
> a^3 + a;
-1
> A.1;
r1
> A.2;
r2
> A.3;
r3
> A.1 eq a;
true
```

It is often useful to use the `Max` parameter with the `Roots` function. Note that in this case A does not have the extra variables found in the previous example.

```
> A := AlgebraicClosure();
> r := Roots(x^3 + x + 1, A: Max := 1);
> A;
Algebraically closed field with 1 variable
Defining relations:
[
    r1^3 + r1 + 1
]
```

One can also create elements by `Sqrt`, etc.

```
> A := AlgebraicClosure();
> sqrt2 := Sqrt(A ! 2);
> cube3 := Root(A!3, 3);
> A;
Algebraically closed field with 2 variables
Defining relations:
[
    r2^3 - 3,
    r1^2 - 2
]
> sqrt2^2;
2
> cube3^3;
3
```

Example H40E2

The n-th Swinnerton-Dyer polynomial is defined to be

$$\prod(x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \cdots \pm \sqrt{p_n}),$$

where p_i is the i-th prime and the product runs over all 2^n possible combinations of $+$ and $-$ signs. Such polynomials lie in $\mathbf{Z}[x]$ and are irreducible over $\mathbf{Z}$. It is very easy to compute them using algebraically closed fields. We simply construct the square roots we need and multiply out the expression, coercing the resulting polynomial to $\mathbf{Z}[x]$.

```
> Z := IntegerRing();
> function SwinnertonDyer(n)
>     P := [2];
>     for i := 2 to n do
>         Append(~P, NextPrime(P[#P]));
>     end for;
>     A := AlgebraicClosure();
>     S := [Sqrt(A ! p): p in P];
>     P<z> := PolynomialRing(A);
>     f := &*[z + &+[t[i]*S[i]: i in [1..n]]: t in CartesianPower({-1, 1}, n)];
>     return PolynomialRing(Z) ! f;
> end function;
> P<x> := PolynomialRing(Z);
> [SwinnertonDyer(i): i in [1..5]];
[
    x^2 - 2,
    x^4 - 10*x^2 + 1,
    x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576,
    x^16 - 136*x^14 + 6476*x^12 - 141912*x^10 + 1513334*x^8 -
        7453176*x^6 + 13950764*x^4 - 5596840*x^2 + 46225,
    x^32 - 448*x^30 + 84864*x^28 - 9028096*x^26 + 602397952*x^24 -
        26625650688*x^22 + 801918722048*x^20 - 16665641517056*x^18 +
        239210760462336*x^16 - 2349014746136576*x^14 +
        15459151516270592*x^12 - 65892492886671360*x^10 +
        172580952324702208*x^8 - 255690851718529024*x^6 +
        183876928237731840*x^4 - 44660812492570624*x^2 +
        2000989041197056
]
```

The Swinnerton-Dyer polynomials yield worse-case inputs for the Berlekamp-Zassenhaus factorization algorithm for polynomials over $\mathbf{Z}$, but they are no longer difficult to factor using van Hoeij's new algorithm (see Example H23E6).
We can even define a simple extension of the Swinnerton-Dyer polynomials. Let $Q = \{q_1, \ldots, q_n\}$ be a set of n distinct primes or negatives of primes. Define:

$$\mathrm{GSD}_Q := \prod(x \pm \sqrt{q_1} \pm \sqrt{q_2} \pm \cdots \pm \sqrt{q_n}),$$

where the product runs over all 2^n possible combinations of $+$ and $-$ signs. Then $\mathrm{GSD}_Q \in \mathbf{Z}[x]$, is irreducible over $\mathbf{Z}$, has degree 2^n, and has at least 2^{n-1} factors mod any prime. A function to compute these polynomials is only a slight variation on the previous function.

```
> function GSD(Q)
>     n := #Q;
>     A := AlgebraicClosure();
>     S := [Sqrt(A ! x): x in Q];
>     z := PolynomialRing(A).1;
>     f := &*[z + &+[t[i]*S[i]: i in [1..n]]: t in CartesianPower({-1, 1}, n)];
>     return PolynomialRing(Z) ! f;
> end function;
```

One can multiply GSD_Q for various Q to construct more reducible polynomials with many modular factors. We first note the effects of changing the sign of the input primes.

```
> GSD([2, 3, 5]);
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
> GSD([-2, -3, -5]);
x^8 + 40*x^6 + 352*x^4 + 960*x^2 + 576
> GSD([-2, 3, 5]);
x^8 - 24*x^6 + 224*x^4 + 960*x^2 + 1600
> GSD([2, -3, 5]);
x^8 - 16*x^6 + 184*x^4 + 960*x^2 + 3600
> GSD([2, 3, -5]);
x^8 + 152*x^4 + 1920*x^2 + 5776
```

We now form a polynomial f which is the product of two degree-64 irreducible polynomials. f has at least 64 factors modulo any prime, but is not difficult to factor using van Hoeij's algorithm.

```
> f := GSD([2, 3, 5, 7, 11, 13])*GSD([-2, -3, -5, -7, -11, -13]);
> Degree(f);
128
> Max([Abs(x): x in Coefficients(f)]);
74356932844713201802276382813294219572861455394629943303351262856515487990904 17
> time L:=Factorization(f);
Time: 9.850
> [Degree(t[1]): t in L];
[ 64, 64 ]
> Max([Abs(x): x in Coefficients(L[1,1])]);
1771080720430629161685158978892152599456 11
```

Example H40E3

This example shows how one can compute Puiseux expansions over an algebraic closure. The `PuiseuxExpansion` function calls the `Roots` function internally as it needs to.

```
> A := AlgebraicClosure();
> S<y> := PuiseuxSeriesRing(A);
> P<x> := PolynomialRing(S);
> f := (x^2 - y^2 - 1)^5 + x*y + 1;
> time S := PuiseuxExpansion(f, 3);
Time: 0.210
> S;
[
    r1*y + (102/2525*r1 - 2/505)*y^3 + O(y^4),
    r2*y + (102/2525*r2 - 2/505)*y^3 + O(y^4),
    r3 + (1/10*r3^2 - 1/10)*y + (-101/1000*r3^7 + 101/200*r3^5 -
        209/200*r3^3 + 26/25*r3)*y^2 + O(y^3),
    r4 + (1/10*r4^2 - 1/10)*y + (-101/1000*r4^7 + 101/200*r4^5 -
        209/200*r4^3 + 26/25*r4)*y^2 + O(y^3),
    r5 + (1/10*r5^2 - 1/10)*y + (-101/1000*r5^7 + 101/200*r5^5 -
        209/200*r5^3 + 26/25*r5)*y^2 + O(y^3),
    r6 + (1/10*r6^2 - 1/10)*y + (-101/1000*r6^7 + 101/200*r6^5 -
        209/200*r6^3 + 26/25*r6)*y^2 + O(y^3),
    r7 + (1/10*r7^2 - 1/10)*y + (-101/1000*r7^7 + 101/200*r7^5 -
        209/200*r7^3 + 26/25*r7)*y^2 + O(y^3),
    r8 + (1/10*r8^2 - 1/10)*y + (-101/1000*r8^7 + 101/200*r8^5 -
        209/200*r8^3 + 26/25*r8)*y^2 + O(y^3),
    r9 + (1/10*r9^2 - 1/10)*y + (-101/1000*r9^7 + 101/200*r9^5 -
        209/200*r9^3 + 26/25*r9)*y^2 + O(y^3),
    r10 + (1/10*r10^2 - 1/10)*y + (-101/1000*r10^7 + 101/200*r10^5 -
        209/200*r10^3 + 26/25*r10)*y^2 + O(y^3)
]
> A;
Algebraically closed field with 10 variables
Defining relations:
[
    r10^8 - 5*r10^6 + 10*r10^4 - 10*r10^2 + 5,
    r9^8 - 5*r9^6 + 10*r9^4 - 10*r9^2 + 5,
    r8^8 - 5*r8^6 + 10*r8^4 - 10*r8^2 + 5,
    r7^8 - 5*r7^6 + 10*r7^4 - 10*r7^2 + 5,
    r6^8 - 5*r6^6 + 10*r6^4 - 10*r6^2 + 5,
    r5^8 - 5*r5^6 + 10*r5^4 - 10*r5^2 + 5,
    r4^8 - 5*r4^6 + 10*r4^4 - 10*r4^2 + 5,
    r3^8 - 5*r3^6 + 10*r3^4 - 10*r3^2 + 5,
    r2^2 + 1/5*r2 - 1,
    r1^2 + 1/5*r1 - 1
]
```

We check that f evaluated at each expansion in S is zero up to the precision.

```
> [Evaluate(f, p): p in S];
[
    O(y^5),
    O(y^5),
    O(y^3),
    O(y^3),
    O(y^3),
    O(y^3),
    O(y^3),
    O(y^3),
    O(y^3),
    O(y^3)
]
```

40.5 Related Structures

`Category(A)` `Parent(A)` `Centre(A)`

`PrimeRing(A)` `PrimeField(A)`

`FieldOfFractions(A)`

40.6 Properties

`BaseField(A)`

Return the base field over which A is defined.

`Rank(A)`

Return the current rank of A, that is, the number of variables which currently define A. This can increase by the construction of new roots, or decrease by pruning (see `Roots` and `Prune` respectively below), so the return value of this function is *variable*.

`Degree(A, v)`

Given an algebraically closed field A of rank r and an integer v in the range $1 \leq v \leq r$, return the current degree of the defining polynomial for variable v. The return value of this function is *variable*, as A may be simplified between invocations, making the defining polynomial for v have smaller degree.

`Degree(A)`

Return the current absolute degree of A, that is, the degree over its base field. This necessitates the simplification of A (see `Simplify` below), so may be very time consuming. The return value varies only when new roots of polynomials over the field are computed, but until then, the return value is invariable (as the field will remain simplified, even if `Prune` or `Absolutize` is called – see below).

`AffineAlgebra(A)`

`QuotientRing(A)`

Return the affine algebra (or multivariate quotient ring) R which currently represents A. The quotient relations of R consist of the defining polynomials A, and one may coerce between A and R, but note that the variable numbers are inverted. The reason for this is that for the system to work, the first root $A.1$ must be the smallest variable with respect to the lexicographical order in the corresponding affine algebra R, so that reductions modulo the Gröbner basis of relations are in the correct form.

Note also that if A changes in any way (whether from simplification or by pruning), then the affine algebra R of course stays the same and will not be comparable with the new form of A if A has a different number of variables than before.

`Ideal(A)`

Return the ideal of defining polynomials currently defining A. This is simply equivalent to `DivisorIdeal(AffineAlgebra(A))`. See the relevant comments for the function `AffineAlgebra`.

40.7 Ring Predicates and Properties

`IsCommutative(A)` `IsUnitary(A)`

`IsFinite(A)` `IsOrdered(A)`

`IsField(A)` `IsEuclideanDomain(A)`

`IsPID(A)` `IsUFD(A)`

`IsDivisionRing(A)` `IsEuclideanRing(A)`

`IsPrincipalIdealRing(A)` `IsDomain(A)`

`A eq B` `A ne B`

`Characteristic(A)`

40.8 Element Operations

40.8.1 Arithmetic Operators

This section lists the basic arithmetic operations available for elements of an algebraically closed field. Elements are always kept in normal form with respect to the defining relations of the field. Computing the inverse of an element may cause a simplification of the field to be performed.

```
+ a    - a
a + b    a - b    a * b    a / b
a ^ k
a +:= b    a -:= b    a *:= b
```

40.8.2 Equality and Membership

40.8.3 Parent and Category

```
Parent(a)    Category(a)
```

40.8.4 Predicates on Ring Elements

```
IsZero(a)
```

Return whether a is the zero element of its field. This is the most difficult of all arithmetic functions for algebraically closed fields! To determine whether a is zero, Magma computes the recursive GCD of a, considered as a polynomial in its highest variable, and the appropriate defining polynomial, to determine the result. If the GCD is non-trivial, then this forces a splitting of the defining polynomial, all elements of the field are reduced, and the original element may now be deemed to be zero (it may not be zero because the cofactor of the GCD may be used to perform the simplification). Despite the fact that simplifications may occur, the return value of this function is *invariable*, and this fact is the most important feature of the whole system, enabling the illusion of a true field to be achieved!

```
IsOne(a)
```

Return whether a is one in its field, which is determined by testing whether $(a - 1)$ is zero. Consequently, a simplification of the field may occur, but the return value is invariable.

```
IsMinusOne(a)
```

Return whether a is minus one in its field, which is determined by testing whether $(a + 1)$ is zero. Consequently, a simplification of the field may occur, but the return value is invariable.

`a eq b`

Return whether $a = b$, which is determined by testing whether $(a - b)$ is zero. Consequently, a simplification of the field may occur, but the return value is invariable.

`a ne b`

`a in A` `a notin A`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsUnit(a)` `IsZeroDivisor(a)` `IsRegular(a)`

`IsIrreducible(a)` `IsPrime(a)`

40.8.5 Minimal Polynomial, Norm and Trace

`MinimalPolynomial(a)`

Return the minimal polynomial of the element a of the field A, relative to the base field of A. This is the unique minimal-degree *irreducible* monic polynomial with coefficients in the base field, having a as a root.

This function works as follows. First the minimal polynomial M of a in the affine algebra corresponding to A is computed. M may be reducible in general, so M is factored, and for each irreducible factor F of M, $F(a)$ is evaluated and it is tested whether this evaluation is zero (using the `IsZero` algorithm). If M is not irreducible, some of these evaluations will cause simplifications of the field, but exactly one of the evaluations will be zero and the corresponding irreducible F is the minimal polynomial of a. Consequently, after F is returned, $F(a)$ will be identically zero so the return value of this function *is invariable.*

Thus the illusion of a true field is sustained by forcing the minimal polynomial of a to be irreducible, by first performing whatever simplifications of A are necessary for this. (In fact, computing minimal polynomials in this way is one method of achieving simplifications.)

`Norm(a)`

Given an element a from an algebraically closed field A, return the absolute norm of a to the base field of A. This is simply computed as $(-1)^{\text{Degree}(M)}$ times the constant coefficient of M, where M is the irreducible minimal polynomial of a returned by `MinimalPolynomial`(a). Consequently, a simplification of the field may occur, but the return value is invariable.

`Trace(a)`

Given an element a from an algebraically closed field A, return the absolute trace of a to the base field of A. This is simply computed as the negation of the coefficient of x^{n-2} in M, where M is the irreducible minimal polynomial of a returned by `MinimalPolynomial`(a). Consequently, a simplification of the field may occur, but the return value is invariable.

`Conjugates(a)`

Given an element a from an algebraically closed field A, return the conjugates of a as a sequence of elements. The conjugates of a are defined to be the roots in A of the minimal polynomial of a, and a is always included. This function is thus simply equivalent to:

```
[t[1] :  t in Roots(MinimalPolynomial(a), A)].
```

(No multiplicities are returned as in the `Roots` function since the minimal polynomial is always squarefree, of course.) As this function first computes the minimal polynomials of a, a simplification of the field may occur, but the return value is invariable.

Example H40E4

We create two elements of an algebraically closed field and note that they are conjugate.

```
> A := AlgebraicClosure();
> x := Sqrt(A!2) + Sqrt(A!-3);
> y := Sqrt(A ! (-1 + 2*Sqrt(A!-6)));
> A;
Algebraically closed field with 4 variables
Defining relations:
[
    r4^2 - 2*r3 + 1,
    r3^2 + 6,
    r2^2 + 3,
    r1^2 - 2
]
> x;
r2 + r1
> y;
r4
> x eq y;
false
> Conjugates(x);
[
    r4,
    -r4,
    r5,
    r6
]
> y in Conjugates(x);
true
```

Of course, x and y are conjugate if and only if they have the same minimal polynomial, which is the case here:

```
> P<z> := PolynomialRing(RationalField());
> MinimalPolynomial(x);
z^4 + 2*z^2 + 25
> MinimalPolynomial(y);
z^4 + 2*z^2 + 25
```

40.9 Simplification

The following procedures allow one to simplify an algebraically closed field so that it is a true field.

`Simplify(A)`

`Partial` BoolElt *Default :* `false`

(Procedure.) Simplify the algebraically closed field A so that the affine algebra which represents it is a true field, modifying A in place. Equivalently, simplify A so that the multivariate polynomial ideal corresponding to the defining polynomials A is maximal.

The procedure first partially simplifies A by calling `MinimalPolynomial` on all variables and sums of two variables of A. This will usually cause many simplifications, since this forces the corresponding minimal polynomials to be irreducible (see `MinimalPolynomial`). The procedure then performs all other necessary simplifications by successively computing absolute representations and factorizing the absolute polynomials which arise. This may be very expensive (in particular, if the final absolute degree is greater than 20), so is only practical for fairly small degrees.

If the parameter `Partial` is set to `true`, then only the partial simplification is performed, which is usually rather fast, and may be sufficient.

`Prune(A)`

Prune the algebraically closed field A by removing useless variables, modifying A in place. That is, for each variable v of A such that its defining polynomial is a linear polynomial, remove v and the corresponding defining polynomial from A, and shift variables higher than v appropriately.

Note that elements of A are kept reduced to normal form with respect to the defining polynomials of A (at least according the user's perception), so for each such v having a linear relation, v cannot occur in any element of A, so the removal of v from A is possible.

40.10 Absolute Field

One may construct an absolute field isomorphic to the current subfield represented by an algebraically closed field. The construction of the absolute field may be very expensive, as it involves factoring polynomials over successive subfields. In fact, it is often the case that the degree of the absolute field is an extremely large integer, so that an absolute field is not practically representable, yet the system may allow one to compute effectively with the original non-absolute presentation.

`AbsoluteAffineAlgebra(A)`

`AbsoluteQuotientRing(A)`

Simplify the algebraically closed field A fully (see `Simplify(A)` above) and then return an absolute field as a univariate affine algebra R which is isomorphic to the current (true) algebraic field represented by A, and also return the isomorphism $f : A \to R$.

`AbsolutePolynomial(A)`

Simplify the algebraically closed field A fully (see `Simplify(A)` above) and then compute an absolute field isomorphic to the current (true) algebraic field represented by A and return the defining polynomial of the absolute field. That is, return a polynomial f such that $K[x]/ < f >$ is isomorphic to A in its current state.

`Absolutize(A)`

Modify the algebraically closed field A in place so that has an absolute presentation. That is, compute an isomorphic absolute field and absolute polynomial f as in `AbsolutePolynomial` and modify A and its elements in place so that A now only has one variable v and corresponding defining polynomial $f(v)$ and the elements of A correspond via the isomorphism to their old representation.

Example H40E5

We show how one can easily compute the number field over which the complete variety of the Cyclic-6 ideal can be defined.
We first create the ideal I over **Q** and compute its variety over A, the algebraic closure of **Q**.

```
> P<a,b,c,d,e,f> := PolynomialRing(RationalField(), 6);
> B := [
>     a + b + c + d + e + f,
>     a*b + b*c + c*d + d*e + e*f + f*a,
>     a*b*c + b*c*d + c*d*e + d*e*f + e*f*a + f*a*b,
>     a*b*c*d + b*c*d*e + c*d*e*f + d*e*f*a + e*f*a*b + f*a*b*c,
>     a*b*c*d*e + b*c*d*e*f + c*d*e*f*a + d*e*f*a*b +
>         e*f*a*b*c + f*a*b*c*d,
>     a*b*c*d*e*f - 1];
> I := ideal<P | B>;
> time Groebner(I);
Time: 1.459
```

```
> A := AlgebraicClosure();
> time V := Variety(I, A);
Time: 4.219
> #V;
156
```

We now notice that there are 28 variables in A and we check that all elements of V satisfy the original polynomials.

```
> Rank(A);
28
> V[1];
<-1, -1, -1, -1, r1 + 4, -r1>
> V[156];
<r28^3 + 2*r28^2*r9 - 2*r9, -r28^3 - 2*r28^2*r9 + 2*r9, r9, -r28, r28, -r9>
> {Evaluate(f, v): v in V, f in B};
{
    0
}
```

We now simplify A to ensure that it represents a true field, and prune away useless variables now having linear defining polynomials.

```
> time Simplify(A);
Time: 3.330
> Prune(A);
> A;
Algebraically closed field with 3 variables
Defining relations:
[
    r3^2 - 1/3*r3*r2*r1 - 5/3*r3*r2 + 2/3*r3*r1 - 2/3*r3 + r2*r1 + 4*r2 + 1,
    r2^2 - r2*r1 - 4*r1 - 1,
    r1^2 + 4*r1 + 1
]
> V[1];
<-1, -1, -1, -1, r1 + 4, -r1>
> V[156];
<2/3*r3*r2*r1 + 7/3*r3*r2 - 1/3*r3*r1 - 2/3*r3 + 5/3*r2*r1 +
    19/3*r2 + 2/3*r1 + 4/3, -2/3*r3*r2*r1 - 7/3*r3*r2 + 1/3*r3*r1
    + 2/3*r3 - 5/3*r2*r1 - 19/3*r2 - 2/3*r1 - 4/3, -4/3*r2*r1 -
    14/3*r2 - 1/3*r1 - 2/3, 2/3*r3*r2*r1 + 7/3*r3*r2 - 1/3*r3*r1
    - 2/3*r3 - 5/3*r2*r1 - 19/3*r2 + 1/3*r1 - 1/3, -2/3*r3*r2*r1
    - 7/3*r3*r2 + 1/3*r3*r1 + 2/3*r3 + 5/3*r2*r1 + 19/3*r2 -
    1/3*r1 + 1/3, 4/3*r2*r1 + 14/3*r2 + 1/3*r1 + 2/3>
```

Finally we compute an absolute polynomial for A, and then modify A in place using `Absolutize` to make A be defined by one polynomial of degree 8.

```
> time AbsolutePolynomial(A);
x^8 + 4*x^6 - 6*x^4 + 4*x^2 + 1
Time: 0.080
```

```
> time Absolutize(A);
Time: 0.259
> A;
Algebraically closed field with 1 variables
Defining relations:
[
    r1^8 + 4*r1^6 - 6*r1^4 + 4*r1^2 + 1
]
> V[1];
<-1, -1, -1, -1, 1/2*r1^6 + 2*r1^4 - 7/2*r1^2 + 3, -1/2*r1^6 -
    2*r1^4 + 7/2*r1^2 + 1>
> V[156];
<r1^7 + 4*r1^5 - 6*r1^3 + 4*r1, -r1^7 - 4*r1^5 + 6*r1^3 - 4*r1,
    -1/4*r1^7 - 3/4*r1^5 + 11/4*r1^3 - 7/4*r1, -r1, r1, 1/4*r1^7
    + 3/4*r1^5 - 11/4*r1^3 + 7/4*r1>
> {Evaluate(f, v): v in V, f in B};
{
    0
}
```

Example H40E6

In this example we compute the splitting field of a certain polynomial of degree 8.
We first set f to a degree-8 polynomial using the database of polynomials with given Galois group. The Galois group has order 16, so we know that the splitting field will have absolute degree 16.

```
> P<x> := PolynomialRing(IntegerRing());
> load galpols;
Loading "/home/magma/libs/galpols/galpols"
> PolynomialWithGaloisGroup(8, 6);
x^8 - 2*x^7 - 9*x^6 + 10*x^5 + 22*x^4 - 14*x^3 - 15*x^2 + 2*x + 1
> f := $1;
> #GaloisGroup(f);
16
```

We next create an algebraic closure A and compute the roots of f over A.

```
> A := AlgebraicClosure();
> r := Roots(f, A);
> #r;
8
> A;
Algebraically closed field with 8 variables
Defining relations:
[
    r8^8 - 2*r8^7 - 9*r8^6 + 10*r8^5 + 22*r8^4 - 14*r8^3 - 15*r8^2 + 2*r8 + 1,
    r7^8 - 2*r7^7 - 9*r7^6 + 10*r7^5 + 22*r7^4 - 14*r7^3 - 15*r7^2 + 2*r7 + 1,
    r6^8 - 2*r6^7 - 9*r6^6 + 10*r6^5 + 22*r6^4 - 14*r6^3 - 15*r6^2 + 2*r6 + 1,
    r5^8 - 2*r5^7 - 9*r5^6 + 10*r5^5 + 22*r5^4 - 14*r5^3 - 15*r5^2 + 2*r5 + 1,
    r4^8 - 2*r4^7 - 9*r4^6 + 10*r4^5 + 22*r4^4 - 14*r4^3 - 15*r4^2 + 2*r4 + 1,
    r3^8 - 2*r3^7 - 9*r3^6 + 10*r3^5 + 22*r3^4 - 14*r3^3 - 15*r3^2 + 2*r3 + 1,
    r2^8 - 2*r2^7 - 9*r2^6 + 10*r2^5 + 22*r2^4 - 14*r2^3 - 15*r2^2 + 2*r2 + 1,
    r1^8 - 2*r1^7 - 9*r1^6 + 10*r1^5 + 22*r1^4 - 14*r1^3 - 15*r1^2 + 2*r1 + 1
]
```

Finally we simplify A. There are defining polynomials of degrees 2 and 8 in the simplified field. The absolute polynomial of degree 16 defines the splitting field of f.

```
> time Simplify(A);
Time: 2.870
> A;
Algebraically closed field with 8 variables
Defining relations:
[
    r8 + 1/2*r3*r1^6 - 2*r3*r1^5 - r3*r1^4 + 8*r3*r1^3 - 2*r3*r1^2 -
        5*r3*r1 - 1/2*r3 + r1^7 - 3/2*r1^6 - 9*r1^5 + 4*r1^4 + 19*r1^3
        - r1^2 - 9*r1 - 5/2,
    r7 - 1/2*r3*r1^6 + 2*r3*r1^5 + r3*r1^4 - 8*r3*r1^3 + 2*r3*r1^2 +
        5*r3*r1 + 1/2*r3,
    r6 + r3 - 3/2*r1^7 + 2*r1^6 + 14*r1^5 - 5*r1^4 - 28*r1^3 + 3*r1^2
        + 19/2*r1,
    r5 - 1/2*r3*r1^6 + 2*r3*r1^5 + r3*r1^4 - 8*r3*r1^3 + 2*r3*r1^2 +
        4*r3*r1 + 1/2*r3 + r1^6 - r1^5 - 9*r1^4 - r1^3 + 14*r1^2 +
        6*r1,
    r4 + 1/2*r3*r1^6 - 2*r3*r1^5 - r3*r1^4 + 8*r3*r1^3 - 2*r3*r1^2 -
        4*r3*r1 - 1/2*r3 - 1,
    r3^2 - 3/2*r3*r1^7 + 2*r3*r1^6 + 14*r3*r1^5 - 5*r3*r1^4 -
        28*r3*r1^3 + 3*r3*r1^2 + 19/2*r3*r1 + 3/2*r1^6 - r1^5 -
        15*r1^4 - 4*r1^3 + 27*r1^2 + 11*r1 - 9/2,
    r2 + 1/2*r1^7 - 3/2*r1^6 - 4*r1^5 + 10*r1^4 + 10*r1^3 - 16*r1^2 -
        11/2*r1 + 3/2,
    r1^8 - 2*r1^7 - 9*r1^6 + 10*r1^5 + 22*r1^4 - 14*r1^3 - 15*r1^2 +
        2*r1 + 1
]
> AbsolutePolynomial(A);
x^16 - 36*x^14 + 488*x^12 - 3186*x^10 + 10920*x^8 - 19804*x^6 + 17801*x^4 -
    6264*x^2 + 64
```

40.11 Bibliography

[**DDD85**] J. Della Dora, C. Dicrescenzo, and D. Duval. About a new method for computing in algebraic number fields. In B.F. Caviness, editor, *Proc. EUROCAL '85*, volume 204 of *LNCS*, pages 289–290, Linz, 1985. Springer.

[**FK02**] Claus Fieker and David R. Kohel, editors. *ANTS V*, volume 2369 of *LNCS*. Springer-Verlag, 2002.

[**Ste02**] Allan Steel. A new scheme for computing with algebraically closed fields. In Fieker and Kohel [FK02], pages 491–505.

[**Ste10**] Allan K. Steel. Computing with algebraically closed fields. *J. Symb. Comput.*, 45(3):342–372, March 2010.

41 RATIONAL FUNCTION FIELDS

Chapter 41
RATIONAL FUNCTION FIELDS

41.1 Introduction

Given a ring R such that there is a greatest-common-divisor algorithm for polynomials over R, MAGMA allows the construction of a rational function field K in any number of indeterminates over R. Such function fields are objects of type `FldFunRat` with elements of type `FldFunRatElt`. The elements of K are fractions whose numerators and denominators lie in the corresponding polynomial ring over R. As for polynomial rings, the different univariate and multivariate cases are distinguished, since the fractions just use the different representations given by the different cases of polynomial rings.

A fraction f lying in a function field K is always *reduced*; this means that the numerator and denominator of f are coprime and the denominator of f is *normalized* (monic over fields and positive over $\mathbf{Z}$). Note that R itself need not be a field. Thus it is possible, for example, to create the rational function field $K = \mathbf{Z}(t)$ which is mathematically equal to $\mathbf{Q}(t)$ of course, but will be represented slightly differently. A fraction in $\mathbf{Q}(t)$ will have a monic denominator (and the coefficients of both the numerator and denominator may be non-integral), while a fraction in $\mathbf{Z}(t)$ will have a positive denominator (and the coefficients of both the numerator and denominator will be integral). Thus the fractions $(3t+2)/(4t-2) \in \mathbf{Z}(t)$ and $((3/4)t+1/2)/(t-1/2) \in \mathbf{Q}(t)$ are equal and are both reduced in their respective fields. It is generally much better to use the domain of integers instead of the field of fractions for the coefficient ring R (so it is better to use $\mathbf{Z}(t)$ instead of $\mathbf{Q}(t)$) since arithmetic is much faster, but the use of a field of fractions for the coefficient ring may be more desirable for output purposes.

41.2 Creation Functions

41.2.1 Creation of Structures

`FunctionField(R)`

`RationalFunctionField(R)`

`Global` BOOLELT *Default :* `true`

Create the field $\mathbf{F}$ of rational functions in 1 indeterminate (consisting of quotients of univariate polynomials) over the integral domain R. The angle bracket notation may be used to assign names to the indeterminates, just as in the case of polynomial rings, e.g.: `K<t> := FunctionField(IntegerRing());`.

By default, the unique *global* univariate function field over R will be returned; if the parameter `Global` is set to `false`, then a non-global univariate function field over R will be returned (to which a separate name for the indeterminate can be assigned).

```
FunctionField(R, r)
RationalFunctionField(R, r)
```

Global BoolElt *Default :* `false`

Create the field **F** of rational functions in r indeterminates over the integral domain R. may be used to assign names to the indeterminates, just as in the case of polynomial rings, e.g.: `K<a,b,c> := FunctionField(IntegerRing(), 3);`.

By default, a *non-global* function field will be returned; if the parameter `Global` is set to `true`, then the unique global function field over R with n variables will be returned. This may be useful in some contexts, but a non-global result is returned by default since one often wishes to have several function fields with the same numbers of variables but with different variable names (and create mappings between them, for example). Explicit coercion is always allowed between function fields having the same number of variables (and suitable base rings), whether they are global or not, and the coercion maps the i-variable of one function field to the i-th variable of the other function field.

```
FieldOfFractions(P)
```

Given a polynomial ring P, return its field of fractions F, consisting of quotients f/g, with $f, g \in P$. The angle bracket notation may be used to assign names to the indeterminates, just as in the case of polynomial rings: `K<t> := FieldOfFractions(P);`. If this function is called more than once for a fixed P, then the identical function field will be returned each time.

41.2.2 Names

```
AssignNames(~F, s)
```

Procedure to change the name of the indeterminates of a function field F. The i-th indeterminate will be given the name of the i-th element of the sequence of strings s (for $1 \le i \le \#s$); the sequence may have length less than the number of indeterminates of F, in which case the remaining indeterminate names remain unchanged.

This procedure only changes the name used in printing the elements of F. It does *not* assign to identifiers corresponding to the strings the indeterminates in F; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies F, it is necessary to have a reference `~F` to F in the call to this function.

```
Name(F, i)
```

Given a function field F, return the i-th indeterminate of F (as an element of F).

41.2.3 Creation of Elements

`F ! [a, b]`

`elt< F | a, b >`

Given the rational function field F (which is the field of fractions of the polynomial ring R), and polynomials a, b in R (with $b \neq 0$), construct the rational function a/b.

`F ! a`

Given the rational function field F as a field of fractions of R, and a polynomial $a \in R$, create the rational function $a = a/1$ in F.

`K . i`

The i-th generator for the field of fractions K of R over the coefficient ring of R.

`One(F)` `Identity(F)`

`Zero(F)` `Representative(F)`

Example H41E1

We create the field of rational functions over the integers in a single variable w.

```
> R<x> := PolynomialRing(Integers());
> F<w> := FieldOfFractions(R);
> F ! x+3;
w + 3
> F ! [ x, x-1 ];
w/(w - 1)
```

41.3 Structure Operations

41.3.1 Related Structures

`IntegerRing(F)`

`RingOfIntegers(F)`

Given the rational function field F this returns the polynomial ring from which F was constructed as its field of fractions.

`BaseRing(F)`

`CoefficientRing(F)`

The coefficient ring of the (ring of integers of) the rational function field F.

`Rank(F)`

The rank (number of indeterminates) of the rational function field F.

`ValuationRing(F)`

Given the rational function field F for which the coefficients come from a field, this returns the valuation ring of F with respect to the valuation given by the degree. This valuation ring consists of those rational functions g/h for which the degree of h is greater than or equal to that of g.

`ValuationRing(F, f)`

Given the rational function field F for which the coefficients come from a field, and an irreducible polynomial f in the ring of integers of F, this returns the valuation ring of F with respect to the valuation associated with f. This valuation ring consists of those rational functions g/h for which f divides g but not h.

`Category(R)` `Parent(R)` `PrimeRing(R)`

41.3.2 Invariants

`Characteristic(F)`

41.3.3 Ring Predicates and Booleans

`IsCommutative(F)` `IsUnitary(F)`

`IsFinite(F)` `IsOrdered(F)`

`IsField(F)` `IsEuclideanDomain(F)`

`IsPID(F)` `IsUFD(F)`

`IsDivisionRing(F)` `IsEuclideanRing(F)`

`IsPrincipalIdealRing(F)` `IsDomain(F)`

`F eq G` `F ne G`

41.3.4 Homomorphisms

In its general form a ring homomorphism taking a function field $R(x_1, \ldots, x_n)$ as domain requires $n + 1$ pieces of information, namely, a map (homomorphism) telling how to map the coefficient ring R together with the images of the n indeterminates.

```
hom< P -> S | f, y1, ..., yn >
```

```
hom< P -> S | y1, ..., yn >
```

Given a function field $F = R(x_1, \ldots, x_n)$, a ring S, a map $f : F \to S$ and n elements $y_1, \ldots, y_n \in S$, create the homomorphism $g : F \to S$ by applying the rules of $g(rx_1^{a_1} \cdots x_n^{a_n}) = f(r)y_1^{a_1} \cdots y_n^{a_n}$ for monomials, linearity for polynomials, i.e., $g(M + N) = g(M) + g(N)$, and division for fractions, i.e., $g(n/d) = g(n)/g(d)$.

The coefficient ring map may be omitted, in which case the coefficients are mapped into S by the unitary homomorphism sending 1_R to 1_S. Also, the images y_i are allowed to be from a structure that allows automatic coercion into S.

Example H41E2

In this example we map $\mathbf{Q}(x, y)$ into the number field $\mathbf{Q}(\sqrt[3]{2}, \sqrt{5})$ by sending x to $\sqrt[3]{2}$ and y to $\sqrt{5}$ and the identity map on the coefficients (which we omit).

```
> Q := RationalField();
> F<x, y> := FunctionField(Q, 2);
> A<a> := PolynomialRing(IntegerRing());
> N<z, w> := NumberField([a^3-2, a^2+5]);
> h := hom< F -> N | z, w >;
> h(x^11*y^3-x+4/5*y-13/4);
-40*w*z^2 - z + 4/5*w - 13/4
> h(x/3);
1/3*z
> h(1/x);
1/2*z^2
> 1/z;
1/2*z^2
```

41.4 Element Operations

41.4.1 Arithmetic

```
+ a        - a
```

```
a + b      a - b      a * b      a / b      a ^ k
```

41.4.2 Equality and Membership

```
a eq b     a ne b
```

```
a in F     a notin F
```

41.4.3 Numerator, Denominator and Degree

`Numerator(f)`

Given a rational function $f \in K$, the field of fractions of R, return the numerator P of $f = P/Q$ as an element of the polynomial ring R.

`Denominator(f)`

Given a rational function $f \in K$, the field of fractions of R, return the denominator Q of $f = P/Q$ as an element of the polynomial ring R.

`Degree(f)`

Given a rational function f in a univariate function field, return the degree of f as an integer (the maximum of the degree of the numerator of f and the degree of the denominator of f).

`TotalDegree(f)`

Given a rational function f in a multivariate function field, return the total degree of f as an integer (the total degree of the numerator of f minus the total degree of the denominator of f).

`WeightedDegree(f)`

Given a rational function f in a multivariate function field, return the weighted degree of f as an integer (the weighted degree of the numerator of f minus the weighted degree of the denominator of f).

41.4.4 Predicates on Ring Elements

`IsZero(a)` `IsOne(a)` `IsMinusOne(a)`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsUnit(a)` `IsZeroDivisor(a)` `IsRegular(a)`

41.4.5 Evaluation

`Evaluate(f, r)`

Given a univariate rational function f in F, return the rational function in F obtained by evaluating the indeterminate in r, which must be from (or coercible into) the coefficient ring of the integers of F.

`Evaluate(f, v, r)`

Given a multivariate rational function f in F, return the rational function in F obtained by evaluating the v-th variable in r, which must be from (or coercible into) the coefficient ring of the integers of F.

41.4.6 Derivative

```
Derivative(f)
```

Given a univariate rational function f, return the first derivative of f with respect to its variable.

```
Derivative(f, k)
```

Given a univariate rational function f, return the k-th derivative of f with respect to its variable. k must be non-negative.

```
Derivative(f, v)
```

Given a multivariate rational function f, return the first derivative of f with respect to variable number v.

```
Derivative(f, v, k)
```

Given a multivariate rational function f, return the k-th derivative of f with respect to variable number v. k must be non-negative.

41.4.7 Partial Fraction Decomposition

```
PartialFractionDecomposition(f)
```

Given a univariate rational function f in $F = K(x)$, return the (unique) complete partial fraction decomposition of f. The decomposition is returned as a (sorted) sequence Q consisting of triples, each of which is of the form $< d, k, n >$ where d is the denominator, k is the multiplicity of the denominator, and n is the corresponding numerator, and also d is irreducible and the degree of n is strictly less than the degree of d. Thus f equals the sum of the $n_t/{d_t}^{k_t}$, where t ranges over the triples contained in Q. If f is improper (the degree of its numerator is greater than or equal to the degree of its denominator), then the first triple of Q will be of the form $< 1, 1, q >$ where q is the quotient of the numerator of f by the denominator of f.

```
SquarefreePartialFractionDecomposition(f)
```

Given a univariate rational function f in $F = K(x)$, return the (unique) complete squarefree partial fraction decomposition of f. The decomposition is returned as a (sorted) sequence Q consisting of triples, each of which is of the form $< d, k, n >$ where d is the denominator, k is the multiplicity of the denominator, and n is the corresponding numerator, and also d is squarefree and the degree of n is strictly less than the degree of d. Thus f equals the sum of the $n_t/{d_t}^{k_t}$, where t ranges over the triples contained in Q. If f is improper (the degree of its numerator is greater than or equal to the degree of its denominator), then the first triple of Q will be of the form $< 1, 1, q >$ where q is the quotient of the numerator of f by the denominator of f.

Example H41E3

We compute the squarefree and complete (irreducible) partial fraction decompositions of a fraction in $Q(t)$.

```
> F<t> := FunctionField(RationalField());
> P<x> := IntegerRing(F);
> f := ((t + 1)^8 - 1) / ((t^3 - 1)*(t + 1)^2*(t^2 - 4)^2);
> SD := SquarefreePartialFractionDecomposition(f);
> SD;
[
    <x^4 + 2*x^3 - x - 2, 1, 467/196*x^3 + 1371/196*x^2 +
        1391/196*x + 234/49>,
    <x^2 - x - 2, 1, -271/196*x + 505/98>,
    <x^2 - x - 2, 2, 271/14*x + 139/7>
]
> // Check appropriate sum equals f:
> &+[F!t[3] / F!t[1]^t[2]: t in SD] eq f;
> D := PartialFractionDecomposition(f);
> D;
[
    <x - 2, 1, -3683/2646>,
    <x - 2, 2, 410/63>,
    <x - 1, 1, 85/36>,
    <x + 1, 1, 1/108>,
    <x + 1, 2, 1/18>,
    <x + 2, 1, 1/18>,
    <x^2 + x + 1, 1, -5/147*x - 8/147>
]
> // Check appropriate sum equals f:
> &+[F!t[3] / F!t[1]^t[2]: t in D] eq f;
true
```

Note that doing the same operation in the function field $Z(t)$ must modify the numerators and denominators to be integral but the result is otherwise the same.

```
> F<t> := FunctionField(IntegerRing());
> P<x> := IntegerRing(F);
> f := ((t + 1)^8 - 1) / ((t^3 - 1)*(t + 1)^2*(t^2 - 4)^2);
> D := PartialFractionDecomposition(f);
> D;
[
    <2646*x - 5292, 1, -3683>,
    <63*x - 126, 2, 25830>,
    <36*x - 36, 1, 85>,
    <108*x + 108, 1, 1>,
    <18*x + 18, 2, 18>,
    <18*x + 36, 1, 1>,
    <147*x^2 + 147*x + 147, 1, -5*x - 8>
]
```

```
> // Check appropriate sum equals f:
> &+[F!t[3] / F!t[1]^t[2]: t in D] eq f;
true
```

Finally, we compute the partial fraction decomposition of a fraction in a function field whose coefficient ring is a multivariate function field.

```
> R<a, b> := FunctionField(IntegerRing(), 2);
> F<t> := FunctionField(R);
> P<x> := IntegerRing(F);
> f := 1 / ((t^2 - a)^2*(t + b)^2*t^3);
> SD := SquarefreePartialFractionDecomposition(f);
> SD;
[
    <x^3 + b*x^2 - a*x - a*b, 1, (-3*a - 2*b^2)/(a^3*b^4)*x^2 +
        (-a - 2*b^2)/(a^3*b^3)*x + (3*a + 3*b^2)/(a^2*b^4)>,
    <x^3 + b*x^2 - a*x - a*b, 2, (a + b^2)/(a^2*b^3)*x^2 +
        1/a^2*x + (-a - b^2)/(a*b^3)>,
    <x, 1, (3*a + 2*b^2)/(a^3*b^4)>,
    <x, 2, -2/(a^2*b^3)>,
    <x, 3, 1/(a^2*b^2)>
]
> // Check appropriate sum equals f:
> &+[F!t[3] / F!t[1]^t[2]: t in SD] eq f;
true
> D := PartialFractionDecomposition(f);
> D;
[
    <x, 1, (3*a + 2*b^2)/(a^3*b^4)>,
    <x, 2, -2/(a^2*b^3)>,
    <x, 3, 1/(a^2*b^2)>,
    <x + b, 1, (-3*a + 7*b^2)/(a^3*b^4 - 3*a^2*b^6 + 3*a*b^8 -
        b^10)>,
    <x + b, 2, -1/(a^2*b^3 - 2*a*b^5 + b^7)>,
    <x^2 - a, 1, (-3*a^2 - 3*a*b^2 + 2*b^4)/(a^6 - 3*a^5*b^2 +
        3*a^4*b^4 - a^3*b^6)*x + (6*a*b - 2*b^3)/(a^5 - 3*a^4*b^2
        + 3*a^3*b^4 - a^2*b^6)>,
    <x^2 - a, 2, (a + b^2)/(a^4 - 2*a^3*b^2 + a^2*b^4)*x -
        2*b/(a^3 - 2*a^2*b^2 + a*b^4)>
]
> // Check appropriate sum equals f:
> &+[F!t[3] / F!t[1]^t[2]: t in D] eq f;
true
```

41.5 Padé-Hermite Approximants

41.5.1 Introduction

A given rational function $F(z) \in k(z)$ over a field k can be written as a power series $f(z)$ in the completion $k((z))$ of $k(z)$ at the place (z). It thus is a good approximation of $f(z)$ in the sense that $F(z)$ is equal to $f(z)$ up to infinite order.

Padé-Hermite approximants deal with the converse. Given a (formal) power series $f(z) \in k[[z]]$ and some non- negative integers n_P and n_Q, find polynomials $P(z), Q(z) \in k[z]$ of degrees at most n_P and n_Q, respectively, such that $P(z)*f(z)-Q(z) = O(z^{n_P+n_Q+1})$ holds. In other words, $P/Q(z)$ is an approximation for $f(z)$ for polynomials P and Q of limited degree in z. In this notation the pair $[P, Q]$ is known as the Padé-Hermite approximant for the power series tuple $(f, -1)$.

The idea of approximating one power series by two polynomials can be extended to approximating several power series at the same time as follows. Consider the vector $\underline{f}^T := (f_1, f_2, \ldots, f_m)^T$ in the m-dimensional vector space $k((z))^m$ over a power series ring. Let $\underline{n} = (n_1, n_2, \ldots, n_m)$ be an m-tuple of non-negative integers. A *Padé-Hermite approximant* of $\underline{f}^T(z)$ of type $\underline{n}$ is a non-zero vector of polynomials $\underline{P} = (P_1, P_2, \ldots, P_m)$ in $k[z]^m$ such that $\underline{P} \cdot \underline{f}^T = O(z^N)$, $N = n_1 + n_2 + \cdots + n_m + m - 1$, is satisfied. A non-trivial Padé-Hermite approximant always exists. The approximant is contained in the sub-space $V_{\underline{f},N} = \{\underline{Q} \in k[z]^m : \underline{Q} \cdot \underline{f}^T = O(z^N)\}$.

The implementation of Padé-Hermite approximants is based on [Der94] and [BL94]. They are implemented in Magma as sequences rather than vectors.

41.5.2 Ordering of Sequences

A Padé-Hermite approximant to some sequence of power series does not have to be unique. They can be ordered according to their maximum degree and their type of sequence. A sequence $\underline{P} = [P_1, P_2, ..., P_m]$ is of degree i if the maximum of the weak degrees of $P_1, P_2, \ldots, P_m$ is i. An extension of the definition maximum degree is when a distortion of non-negative integers $\underline{d} = [d_1, d_2, \ldots, d_m]$ on the degrees is allowed. In this case the maximum degree is defined as the maximum of $\deg P_i - d_i$. The type of $\underline{P}$ identifies the last P_i in $\underline{P}$ whose weak degree equals the maximum degree of $\underline{P}$. Again a distortion on the degrees is allowed.

`MaximumDegree(f)`

`Distortion` SeqEnum *Default :* []

MaximumDegree returns the degree of a sequence of polynomials or power series, defined as the maximum of the degrees of $f[i] - d[i]$, where d is the distortion. The value -infinity is returned in the case that f is weakly equal to the zero-sequence.

Example H41E4

```
> S<u> := PowerSeriesRing(Rationals());
> f := [u+u^2, 2+u^2+u^3,0];
> MaximumDegree(f);
3
> MaximumDegree(f:Distortion:=[]);
3
> MaximumDegree(f:Distortion:=[0,2,1]);
2
> MaximumDegree([S|0,0]);
-Infinity
> MaximumDegree([O(u)]);
-Infinity
```

`TypeOfSequence(f)`

`Distortion` SeqEnum *Default :* []

Returns the highest index i for those $f[i]$ whose (distorted) degree is weakly equal to the maximum of the degrees of all entries. The second integer returned is the maximum degree of the sequence.

Example H41E5

```
> S<u> := PowerSeriesRing(Rationals());
> f := [u+u^2, 2+u^2+u^3,0];
> TypeOfSequence(f);
2 3
> TypeOfSequence(f:Distortion:=[]);
2 3
> TypeOfSequence(f:Distortion:=[0,2,1]);
1 2
> TypeOfSequence([S|0,0]);
2 -Infinity
```

The Padé-Hermite approximant of $\underline{f}$ can be seen as an element of the free module $k[z]^m$ of rank m, or as an element of the sub-space $V_{\underline{f},N}$.

A free sub-model $V \subset k[z]^m$ is generated by m polynomial vectors $\underline{Q}_i$, $1 = 1, 2, \ldots, m$, such that $\underline{Q}_1(z), \underline{Q}_2(z), \ldots, \underline{Q}_m(z)$ form a minimal vector sequence of V. Such a sequence is defined as a sequence S of m vectors in V, such that $S[i]$ is a non-trivial polynomial vector in V of minimal degree of type i, for $i = 1, 2, \ldots, m$. A minimal vector sequence is not unique.

Two variations on a minimal vector sequence are implemented. The first allows a distortion as an attribute. The sequence is the based on the distorted maximal degree. The second attribute sets the positive power p of z in each of the $\underline{Q}_i$ as follows. Instead of considering $\underline{Q}_i(z)$ for each i, one considers $\underline{Q}_i(z^p)$.

`MinimalVectorSequence(f,n)`

Distortion	SeqEnum	*Default :* []
Power	RngIntElt	*Default :* 1

A minimal sequence of vectors $\underline{Q}_1, \underline{Q}_2, \ldots, \underline{Q}_m$ with respect to the sequence f of length m whose entries are polynomials or power series. The order of $\underline{Q}_i \cdot f$ is at least m.

Example H41E6

```
> S<u> := PowerSeriesRing(Rationals());
> f := [u+u^2, 2+u^2+u^3];
> seq := MinimalVectorSequence(f, 2);
> seq;
[
    (u 0),
    (   -1 1/2*u)
]
> sums := [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
> sums;
[
    u^2 + u^3,
    -u^2 + 1/2*u^3 + 1/2*u^4
]
> seq := MinimalVectorSequence(f,3);
> #seq eq 2,  seq[1], seq[2];
true (u^2   0)
(-1 + u  1/2*u)
```

Example H41E7

```
> L<x> := PolynomialRing(Rationals());
> f := [1+x, 3-x^2, 5+x+x^3-x^5];
> seq := MinimalVectorSequence(f, 10);
> seq[1];
(-2*x^2 + 6   -2*x - 2          0)
> sums := [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
> sums;
[
    0,
    0,
    x^10
]
```

Example H41E8

```
> S<u> := PowerSeriesRing(Rationals());
> f := [2*u^4,2+u^3+u^6];
> seq := MinimalVectorSequence(f, 10);
> seq;
[
    (1/2*u^6       0),
    (-1/2 - 1/4*u^3        1/2*u^4)
]
> sums := [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
> sums;
[
    u^10,
    1/2*u^10
]
```

Example H41E9

```
> S<u> := PowerSeriesRing(Rationals());
> f :=  [1+u-7*u^2, 6-3*u+1/2*u^2-u^3, 5-u+u^2];
> seq := MinimalVectorSequence(f, 5);
>  [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
[
    0,
    u^5,
    0
]
> seq := MinimalVectorSequence(f, 5:Distortion:=[2,0,1]);
> [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
[
    -7/2*u^5,
    u^5,
    0
]
> p:=2;
> seq := MinimalVectorSequence(f, 5:Distortion:=[2,0,1], Power:=p);
> sums := [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
> sums;
> mp:= map<S->S|  x :-> (IsWeaklyZero(x) select 0
>     else  &+([Coefficient(x,i)*(S.1)^(p*i) :  i in Exponents(x)]))
>       + (ISA(Type(v),RngIntElt) select O((S.1)^(p*v))
>   else S!0 where v := AbsolutePrecision(x))>;
> sums := [&+([mp(Q[i])*f[i]: i in [1..#f]]) : Q in seq];
> sums;
[
    u^6 + u^7 - 7*u^8,
    u^5 - 23/3*u^6,
    -5/3*u^5 + 35/3*u^6
]
```

41.5.3 Approximants

The Padé-Hermite approximant of type $\underline{d} = [d_1, d_2, \ldots, d_m]$ with respect to the tuple $\underline{f}^T \in k((z))^m$ is an element of the space $V_{\underline{f},N} = \{\underline{Q} \in k[z]^m : \underline{Q} \cdot \underline{f}^T = O(z^N)\}$ for N equal to $d_1 + d_2 + \cdots + d_m + m - 1$. This space is generated by the vectors in the minimal vector sequence with respect to $\underline{f}$ with distortion $\underline{d}$. The routine `PadeHermiteApproximant` returns one that is smallest with respect to the degree on sequences. The input sequence $\underline{f}$ must be a sequence of polynomial ring elements, or be a power series sequence. While the Padé Hermite approximants theoretically are polynomials, MAGMA returns them as elements of the same ring the entries of $\underline{f}$ are contained in.

```
PadeHermiteApproximant(f,d)
```

`Power` RNGINTELT *Default :* 1

Returns a Padé-Hermite form $\underline{P}$ of f with distortion d, smallest with respect to the degree on sequences, and the corresponding minimal vector sequence. The third argument returned is the order in the order term of $\underline{P} \cdot f$.

Example H41E10

This example can be found on page 813 in [BL94].

```
> S<u> := PowerSeriesRing(Rationals());
> f := [1,u,u/(1-u^4)+u^10+O(u^16),u/(1+u^4)+u^12+O(u^16)];
> pade, padebasis, ord := PadeHermiteApproximant(f,[2,2,2,2]);
> pade, ord;
( u -1  0  0)
11
> BaseRing(Parent(pade)) eq S;
true
> MinimalVectorSequence(f,10);
[
    ( u -1  0  0),
    (   0  u^4 -1/2  1/2),
    (          0    1 - u^4 -1/2 + u^4       -1/2),
    (    0    -u 1/2*u 1/2*u)
]
>
> p := 2;
> seq := MinimalVectorSequence(f,10: Distortion :=[2,2,2,2],Power := p);
> seq;
[
    (u^5   0   0   0),
    (   0  u^2 -1/2  1/2),
    (         0    1 - u^2 -1/2 + u^2       -1/2),
    (    0    -u 1/2*u 1/2*u)
]
> mp:= map<S->S|  x :-> (IsWeaklyZero(x) select 0
>     else  &+([Coefficient(x,i)*(S.1)^(p*i) :  i in Exponents(x)]))
>       + (ISA(Type(v),RngIntElt) select O((S.1)^(p*v))
>   else S!0 where v := AbsolutePrecision(x))>;
> sums := [&+([mp(Q[i])*f[i]: i in [1..#f]]) : Q in seq];
> [Valuation(v) : v in sums];
[ 10, 10, 10, 11 ]
> sums;
[
    u^10,
    -1/2*u^10 + 1/2*u^12 - u^13 + O(u^16),
    -1/2*u^10 - 1/2*u^12 + u^13 + u^14 + O(u^16),
    u^11 + 1/2*u^12 + 1/2*u^14 + O(u^18)
]
```

Example H41E11

This example covers the example on page 815 in [BL94].

```
> S<u> := PowerSeriesRing(Rationals());
> f := [1,u, -1-u^4-2*u^8+u^10+u^11-u^12+O(u^16),-u-u^5-u^9-u^14-u^15+O(u^16)];
> dist:=[2,2,3,3];
> seq := MinimalVectorSequence(f,13:Distortion:=dist);
> pade, padebasis, ord :=  PadeHermiteApproximant(f,dist);
> pade, ord;
(-u  1  0  0)
13
> padebasis;
[
    (-u  1  0  0),
    (1/2*u -1/2*u^4 1/2*u + 1/2*u^3 + 1/2*u^4 -1/2*u^2 - 1/2*u^3 - u^4),
    (-1/2 1/2*u^3 -1/2 - 1/2*u^2 - 1/2*u^3 - u^4 1/2*u + 1/2*u^2 + 2*u^3),
    (      -u        0        0 -1 + u^4)
]
> padebasis eq seq;
true
> [[Valuation(w[i]): i in [1..Degree(w)]] : w in seq];
[
    [ 1, 0, Infinity, Infinity ],
    [ 1, 4, 1, 2 ],
    [ 0, 3, 0, 1 ],
    [ 1, Infinity, Infinity, 0 ]
]
> [[MaximumDegree([w[i]])-dist[i]: i in [1..Degree(w)] ] : w in seq];
```

```
[
    [ -1, -2, -Infinity, -Infinity ],
    [ -1, 2, 1, 1 ],
    [ -2, 1, 1, 0 ],
    [ -1, -Infinity, -Infinity, 1 ]
]
> p:=2;
> seq := MinimalVectorSequence(f,12:Distortion:=dist,Power:=p);
> seq;
[
    (  u - u^3         0 u - 2*u^3         0),
    (-1 - u + u^2 -u^3 -1 - u + 2*u^2 + u^3 -u^3),
    (u + u^2 - u^3 0 u + u^2 - 2*u^3 - u^4 0),
    (      0       1       0 1 - u^2)
]
> seq[1]-seq[3];
(      -u^2          0 -u^2 + u^4          0)
> mp:= map<S->S|  x :-> (IsWeaklyZero(x) select 0
>     else  &+([Coefficient(x,i)*(S.1)^(p*i) :  i in Exponents(x)]))
>       + (ISA(Type(v),RngIntElt) select O((S.1)^(p*v))
>   else S!0 where v := AbsolutePrecision(x))>;
>  [Valuation(&+([mp(Q[i])*f[i]: i in [1..#f]])) : Q in seq];
[ 12, 12, 13, 13 ]
```

Example H41E12

This example considers the example on page 816 in [BL94].

```
> S<u> := PowerSeriesRing(Rationals());
> f := [1,u,-1-u^4-2*u^8+u^10+O(u^12),-u-u^5-u^9+u^10+O(u^12)];
> dist := [2,2,3,3];
> seq := MinimalVectorSequence(f,12: Distortion := dist);
> [[MaximumDegree([w[i]])-dist[i]: i in [1..Degree(w)] ] : w in seq];
[
    [ -1, -2, -Infinity, -Infinity ],
    [ -2, 1, 0, 0 ],
    [ -Infinity, -Infinity, 1, 0 ],
    [ -1, -Infinity, 0, 1 ]
]
> [Valuation(&+([(Q[i])*f[i]: i in [1..#f]]) ) : Q in seq];
[ Infinity, 12, 12, 12 ]
> [MaximumDegree([ &+([(Q[i])*f[i]: i in [1..#f]]) ]) : Q in seq];
[ -Infinity, -Infinity, 14, -Infinity ]
> PadeHermiteApproximant(f,[2,2,3,3]);
> p := 2;
> seq := MinimalVectorSequence(f,12:Distortion:=[2,2,3,3],Power:=p);
> seq;
[
    (  1 - u^2        -1 1 - 2*u^2  -1 + u^2),
    (   0 -u^4    0 -u^4),
    (      -u       -1 -u + u^3 -1 + u^2),
    (      0      u      0 u - u^3)
]
> [[MaximumDegree([w[i]])-dist[i]: i in [1..Degree(w)] ] : w in seq];
[
    [ 0, -2, -1, -1 ],
    [ -Infinity, 2, -Infinity, 1 ],
    [ -1, -2, 0, -1 ],
    [ -Infinity, -1, -Infinity, 0 ]
]
> mp:= map<S->S|  x :-> (IsWeaklyZero(x) select 0
>     else  &+([Coefficient(x,i)*(S.1)^(p*i) :  i in Exponents(x)]))
>       + (ISA(Type(v),RngIntElt) select O((S.1)^(p*v))
>   else S!0 where v := AbsolutePrecision(x))>;
>
> [Valuation(&+([mp(Q[i])*f[i]: i in [1..#f]])) : Q in seq];
[ 12, 13, 12, 12 ]
```

A variant of the Padé-Hermite approximant is when the exponent in the order term is set rather that the type of the sequence. It is also possible to let $\underline{f}$ be a sequence such that its entries themselves are vectors of polynomials or power series.

`PadeHermiteApproximant(f,m)`

`Power` RngIntElt *Default :* 1

Returns a Padé-Hermite form of minimal degree in the corresponding minimal vector sequence, such that its inproduct with f has order at least m. The second argument returned is the corresponding minimal vector sequence.

Example H41E13

This example can be found on page 813 in [BL94].

```
> S<u> := PowerSeriesRing(Rationals());
> f := [Vector([1]), Vector([u])];
> pade, seq := PadeHermiteApproximant(f,3);
Calculating the Pade'-Hermite approximant for the sequence [
    1,
    u
]
with order term 3 and power 1 .
> pade;
( u -1)
> seq;
[
    ( u -1),
    (  0 u^2)
]
> mat := Matrix([Eltseq(v): v in f]);
> pade*mat;
(0)
> PadeHermiteApproximant([1,u],5);
( u -1)
[
    ( u -1),
    (  0 u^4)
]
> PadeHermiteApproximant(f,3:Power:=2);
> g:= [Vector([1,0,0]), Vector([0,1,0]), Vector([1+u,2+u^2,u^3])];
> pade := PadeHermiteApproximant(g,5);
Calculating the Pade'-Hermite approximant for the sequence [
    1,
    u,
    1 + 2*u + u^3 + u^7 + u^11
]
with order term 15 and power 3 .
> pade;
(-u^3 - u^4     -2*u^3        u^3)
> pade*Matrix([Eltseq(v): v in g]);
(  0 u^5 u^6)
```

Example H41E14

This example considers Padé-Hermite approximants for some series that have non-trivial power series expansions.

```
> S<u> := PowerSeriesRing(Rationals());
> f := [Sin(u), Cos(u), Exp(u)];
> [Valuation(f[i]) : i in [1..#f]], [Degree(f[i]) : i in [1..#f]];
[ 1, 0, 0 ]
[ 19, 20, 20 ]
> [AbsolutePrecision(f[i]) : i in [1..#f]];
[ 21, 22, 21 ]
> dist := [3,2,5];
> pade, seq, ord := PadeHermiteApproximant(f,dist);
> 1/420*pade;
(-1275 - 255*u + 45*u^2 + 5*u^3 120 + 495*u + 75*u^2 -120 + 900*u - 600*u^2 +
    160*u^3 - 20*u^4 + u^5)
> ord eq &+(dist)+#f-1, ord;
true 12
> [Degree(pade[i]) : i in [1..Degree(pade)]];
[ 3, 2, 5 ]
> g:= [Cos(2*u)*(u+1)+3,Cos(u)^2+u*Cos(u)+1,Cos(2*u)+1,Cos(u)];
> pade, basis := PadeHermiteApproximant(g,20);
> 131/75880*pade;
(         2    -4 + 2*u        -3*u 4*u - 2*u^2)
> h := [ 1+u^2-u^7+u^12, Sin(u), Exp(u) ];
> dist:=[3,1,2];
> seq := MinimalVectorSequence(h,8:Distortion := dist);
> sums := [&+([Q[i]*h[i]: i in [1..#f]]) : Q in seq];
> [Valuation(s) : s in sums];
[ 8, 8, 8 ]
> [[MaximumDegree([w[i]]): i in [1..Degree(w)] ] : w in seq];
[
    [ 4, 1, 2 ],
    [ 4, 2, 2 ],
    [ 3, 1, 2 ]
]
>  [[MaximumDegree([w[i]])-dist[i]: i in [1..Degree(w)] ] : w in seq];
[
    [ 1, 0, 0 ],
    [ 1, 1, 0 ],
    [ 0, 0, 0 ]
]
```

41.6 Bibliography

[**BL94**] Bernhard Beckermann and George Labahn. A uniform approach for the fast computation of matrix-type Padé approximants. *SIAM J. Matrix Anal. Appl.*, 15(3): 804–823, 1994.

[**Der94**] Harm Derksen. An algorithm to compute generalized Padé-Hermite Forms. Technical Report 9403, Department of Mathemtaics, Catholic University Nijmegen, jan 1994.

42 ALGEBRAIC FUNCTION FIELDS

Chapter 42
ALGEBRAIC FUNCTION FIELDS

42.1 Introduction

This version of Magma is based on the KANT Version 4 system for algebraic function field computations. Special functions for rational function fields are described in Chapter `FldFunRat`.

An algebraic function field F/k (in one variable) over a field k is a field extension F of k such that F is a finite field extension of $k(x)$ for an element $x \in F$ which is transcendental over k. As a Magma object it is of type `FldFun` with elements of type `FldFunElt`. For perfect k it is always possible to choose $x \in F$ so that $F/k(x)$ is also separable. For such x there exists a primitive element $\alpha \in F$ with $F = k(x, \alpha)$ where α is a root of an irreducible, separable polynomial in $k(x)[y]$.

42.1.1 Representations of Fields

All function fields in Magma can be represented as described above, i.e. as $k(x, \alpha)$ where x is transcendental and α is algebraic. This is the representation which has the most functionality.

Alternatively, one may wish to consider a function field as a combined transcendental and algebraic extension of its constant field, more like a curve. Such a field would be a quotient of $k[x, y]$. Fields in this representation do not have orders.

Algebraic function fields may be extended to create relative finite extensions of $k(x)$ like $F = k(x, \alpha_1, \ldots, \alpha_n)$. The functionality described in this chapter which is not available for these relative extensions is that involving series rings, galois groups and subfields.

It is also possible to make non–simple extensions where more than one root of a polynomial is added at each step by extending by several polynomials. These extensions have the same functionality as the relative finite extensions, except that primitive elements and some functions involving differentials are not available.

Function fields represented as finite extensions may have orders. Some orders will have a basis different to that of their function fields. Orders may have a field of fractions. A field of fractions of an order is isomorphic to the function field of the order, however its elements are represented with respect to the basis of the order. So there are 3 different types of rings covered in this chapter, function fields (`FldFun`), orders of function fields (`RngFunOrd`) and fields of fractions of orders of function fields (`FldFunOrd`). Each ring type has its own corresponding element type.

42.2 Creation of Algebraic Function Fields and their Orders

42.2.1 Creation of Algebraic Function Fields

```
ext< K | f >
```

`FunctionField(f :`*parameters*`)`

`Check`	BoolElt	*Default :* `true`
`Global`	BoolElt	*Default :* `true`

Let k be a field and $K = k(x)$ or $K = k(x, \alpha_1, \ldots, \alpha_r)$ some finite extension of $k(x)$. Given an irreducible and separable polynomial $f \in K[y]$ of degree greater than zero with coefficients within K, create the algebraic function field $F = K[y]/\langle f \rangle = k(x, \alpha_1, \ldots, \alpha_r, \alpha)$ obtained by adjoining a root α of f to K. F will be viewed as a (finite) extension of K. The polynomial f is also allowed to be $\in k[x][y]$.

The optional parameter `Check` may be used to prevent some conditions from being tested. The default is `Check := true`, so that f is verified to be irreducible and separable. The optional parameter `Global` may be used to allow another copy of the field to be returned if it is set to `false`, otherwise if a field has already been constructed using f over K and has not been deleted then the existing field will be returned.

The angle bracket notation may be used to assign the root α to an identifier: `F<a> := FunctionField(f)`.

`FunctionField(f :`*parameters*`)`

`Check`	BoolElt	*Default :* `true`
`Global`	BoolElt	*Default :* `true`

Let k be a field. Given an irreducible polynomial $f \in k[x, y]$ of degree greater than zero, create the algebraic function field F which is the field of fractions of $k[x, y]/\langle f \rangle$. The polynomial f must be separable in at least one variable. F will be viewed as (infinite) extension of k.

The optional parameter `Check` may be used to prevent some conditions from being tested. The default is `Check := true`, so that f is verified to be irreducible and separable in at least one variable. The optional parameter `Global` may be used to allow another copy of the field to be returned if it is set to `false`, otherwise if a field has already been constructed using f over K and has not been deleted then the existing field will be returned.

The angle bracket notation may be used to assign the images of x, y in F to identifiers: `F<a, b> := FunctionField(f)`.

```
FunctionField(S)
```

`Check` BoolElt *Default :* `true`

Return the function field F whose defining polynomials are the polynomials in the sequence S. If `Check` is set to `false` then it will not be checked the polynomials actually define a field.

```
FunctionField(S)
```

Return the function field F whose defining polynomials are the polynomials in the sequence S. This field F will be represented as an infinite degree extension of the coefficient field of the polynomials in S.

```
HermitianFunctionField(p, d)
HermitianFunctionField(q)
```

Create the Hermitian function field $F = \mathbf{F}_{q^2}(x, \alpha)$ defined by $\alpha^q + \alpha = x^{q+1}$, where q is the d-th power of the prime number p.

`sub< F | S >`

`sub< F |` $s_1, \ldots, s_r$ `>`

The subfield of the function field F containing the elements in the sequence S or the elements s_i.

```
AssignNames(~F, s)
AssignNames(~a, s)
```

Procedure to change the name of the generating element(s) in the function field F (a in F) to the contents of the sequence of strings s, which must have length 1 or 2 in this case.

This procedure only changes the name(s) used in printing the elements of F. It does *not* assign to any identifier(s) the value(s) of the generator(s) in F; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies F, it is necessary to have a reference `~F` to F (or a) in the call to this function.

```
FunctionField(R)
```

`Global` BoolElt *Default :* `true`

`Type` Cat *Default :* `FldFunRat`

Return the rational function field over R in one variable. If `Global` is `false` then create a new copy of the field, otherwise reuse any globally created field which already exists. If `Type` is `FldFun` create the field as an algebraic function field, otherwise create as a rational function field.

Example H42E1

Let $\mathbf{F}_5$ be the finite field of five elements. To create the function field extension $\mathbf{F}_5(x, \alpha)/\mathbf{F}_5(x)$, where α satisfies

$$\alpha^2 = 1/x,$$

one may proceed in the following, equivalent ways:

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> F<alpha> := FunctionField(y^2 - 1/x);
> F;
Algebraic function field defined over Univariate rational function field over
GF(5)
Variables: x by
y^2 + 4/x
```

or

```
> R<x> := PolynomialRing(GF(5));
> P<y> := PolynomialRing(R);
> F<alpha> := FunctionField(x*y^2 - 1);
> F;
Algebraic function field defined over Univariate rational function field over
GF(5)
Variables: x by
x*y^2 + 4
```

or

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> F<alpha> := ext< R | y^2 - 1/x >;
> F;
Algebraic function field defined over Univariate rational function field over
GF(5)
Variables: x by
y^2 + 4/x
```

Example H42E2

An extension of F may be created as follows.

```
> R<y> := PolynomialRing(F);
> FF<beta> := FunctionField(y^3 - x/alpha : Check := false);
> FF;
Algebraic function field defined over F by
y^3 + 4*x^2*alpha
```

Example H42E3

To create a non–simple extension:

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> FF<alpha, beta> := FunctionField([y^2 - 1/x, y^3 + x]);
> FF;
Algebraic function field defined over Univariate rational function field over
GF(5) by
y^2 + 4/x
y^3 + x
```

or

```
> P<y> := PolynomialRing(F);
> FF<beta, gamma> := FunctionField([y^2 - x/alpha, y^3 + x]);
> FF;
Algebraic function field defined over F by
y^2 + 4*x^2*alpha
y^3 + x
```

Example H42E4

The creation of an Hermitian function field:

```
> F := HermitianFunctionField(9);
> F;
Algebraic function field defined over GF(3^4) by
y^9 + y + 2*x^10
```

42.2.2 Creation of Orders of Algebraic Function Fields

Equation orders, maximal orders and other orders of algebraic function fields can be created.

`EquationOrderFinite(F)`

Create the 'finite' equation order of the function field $F/k(x, \alpha_1, \ldots, \alpha_r)$, i.e. $k[x, d_1\alpha_1, \ldots, d_r\alpha_r, d\alpha]$ where $d_j, d \in k[x]$ is chosen such that $d_j\alpha_j, d\alpha$ are integral over $k[x]$.

`MaximalOrderFinite(F)`

Create the 'finite' maximal order of the function field $F/k(x, \alpha_1, \ldots, \alpha_r)$. This is the integral closure of $k[x, d_1\alpha_1, \ldots, d_r\alpha_r]$ in F.

`EquationOrderInfinite(F)`

Create the 'infinite' equation order of the function field $F/k(x, \alpha_1, \ldots, \alpha_r)$, i.e. $o_\infty[\alpha_1, \ldots, \alpha_r, \beta]$ where o_∞ denotes the valuation ring of the degree valuation in $k(x)$ and β is a primitive element of $F/k(x, \alpha_1, \ldots, \alpha_r)$ which is integral over o_∞.

`MaximalOrderInfinite(F)`

Create the 'infinite' maximal order of the function field $F/k(x, \alpha_1, \ldots, \alpha_r)$. This is the integral closure of o_∞ in F.

`IntegralClosure(R, F)`

The integral closure of the subring R of the function field F in itself.

`EquationOrder(O)`

The equation order of the order O. An order whose basis is a transformation of that of O and is a power basis.

`MaximalOrder(O)`

`Discriminant`	Any	*Default :*
`Ramification`	SeqEnum	*Default :*
`Al`	MonStgElt	*Default : "Auto"*
Verbose	`MaximalOrder`	*Maximum :* 5

The maximal order of the order O of an algebraic function field.

If O is a radical (pure) extension then specific code is used to calculate each p-maximal order, rather than the Round 2 method. In this case we can compute a pseudo basis for the p-maximal orders knowing only the valuation of the constant coefficient of the defining polynomial at p [Sut12].

If O is an Artin–Schreier extension then the maximal order can be computed directly without computing the p-maximal orders. The proof of Proposition III.7.8 of [Sti93] gives us a start on some elements which are a basis for the maximal order [Sut13].

If the `Discriminant` or `Ramification` parameters are supplied an algorithm ([Bj94], Theorems 1.2 and 7.6) which can compute the maximal order given the discriminant of the maximal order will be used. `Discriminant` must be an element of the coefficient ring of O if O is a non relative order and must be an ideal of O if O is a relative order. `Ramification` must contain elements of the coefficient ring if O is a non relative order and must contain ideals of O if O is a relative order. The ramification sequence is taken to contain prime factors of the discriminant. Only one of these parameters can be specified and if one of them is then `Al` cannot be specified. Otherwise `Al` may be set to `"Round2"` to avoid using the algorithms for the special cases above.

`SetOrderMaximal(O, b)`

Set the order O of a function field to be maximal if b is `true` and to be non–maximal if b is `false`.

`ext< O | f >`

`Check`	Bool	*Default :* `true`

The order O with a root of f adjoined.

Example H42E5

Creation of orders is shown below.

```
> PR<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(PR);
> FR1<a> := FunctionField(y^3 - x*y^2 + y + x^4);
> P<y> := PolynomialRing(FR1);
> FR2<c> := FunctionField(y^2 + y - a/x^5);
> EFR1F := EquationOrderFinite(FR1);
> MFR1F := MaximalOrderFinite(FR1);
> EFR1I := EquationOrderInfinite(FR1);
> MFR1I := MaximalOrderInfinite(FR1);
> EFR2F := EquationOrderFinite(FR2);
> MFR2F := MaximalOrderFinite(FR2);
> EFR2I := EquationOrderInfinite(FR2);
> MFR2I := MaximalOrderInfinite(FR2);
>  MaximalOrder(EFR2I);
>> MaximalOrder(EFR2I);
               ^
Runtime error in 'MaximalOrder': Order must be defined over a maximal order
> MFR2I;
Maximal Order of FR2 over MFR1I
> P<y> := PolynomialRing(FR1);
> MaximalOrder(ext<MFR1F | y^2 + y - a*x^5>); MFR2F;
Maximal Equation Order of Algebraic function field defined over FR1 by
y^2 + y - x^5*a over EFR1F
Maximal Order of FR2 over EFR1F
> MaximalOrder(ext<MFR1I | y^2 - 1/a>);
Maximal Order of Algebraic function field defined over FR1 by
y^2 + 1/x^4*a^2 - 1/x^3*a + 1/x^4 over MFR1I
```

Example H42E6

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> f := y^3 + (4*x^3 + 4*x^2 + 2*x + 2)*y^2 + (3*x + 3)*y + 2;
> F<alpha> := FunctionField(f);
> IntegralClosure(R, F);
```

```
Algebraic function field defined over GF(5) by
y^3 + (4*x^3 + 4*x^2 + 2*x + 2)*y^2 + (3*x + 3)*y + 2
> IntegralClosure(PolynomialRing(GF(5)), F);
Maximal Order of F over Univariate Polynomial Ring in x over GF(5)
> IntegralClosure(ValuationRing(R), F);
Maximal Order of F over Valuation ring of Rational function field of
rank 1 over GF(5)
Variables: x with generator 1/x
```

`Order(O, T, d)`

Check	BoolElt	*Default :* true

Create the order whose basis is that of the order O multiplied by the matrix T over the coefficient ring of O divided by the scalar d. If the parameter `Check` is set to `false` then it will not be checked that the result is actually an order (potentially expensive).

`Order(O, M)`

Check	BoolElt	*Default :* true

Create the order whose basis is that of the order O multiplied by the dedekind module M. If the parameter `Check` is set to `false` then it will not be checked that the result is actually an order (potentially expensive).

`Order(O, S)`

Verify	BoolElt	*Default :* true
Order	BoolElt	*Default :* false

Given a sequence S of elements in an algebraic function field F create the minimal order R of F which contains all elements of S.

The order O may be an order of F which will be used as the suborder of R, in which case its coefficient ring should be maximal, or O may be a maximal order of the coefficient field of F.

If `Verify` is `true`, it is verified that the elements of S are integral algebraic numbers. This can be a lengthy process if the field is of large degree.

Setting `Order` to `true`, when the order is large will avoid verifying that the module generated by the elements of S is closed under multiplication. By default, products of the generators will be added until the module is closed under multiplication.

`Simplify(O)`

Return the order O as a direct transformation of its equation order, instead of a composition of transformations.

```
O1 + O2
```

The smallest common over order of $O1$ and $O2$ where $O1$ and $O2$ have the same equation order.

```
O1 meet O2
```

The intersection of orders $O1$ and $O2$ which must have the same equation order.

```
AsExtensionOf(O1, O2)
```

Return the order $O1$ as an transformation of the order $O2$ where $O1$ and $O2$ have the same coefficient ring.

Example H42E7

Some of the above order creations are shown below.

```
> P<x> := PolynomialRing(GF(5));
> P<y> := PolynomialRing(P);
> F<a> := FunctionField(y^3 - x^4);
> O := Order(EquationOrderFinite(F), MatrixAlgebra(Parent(x), 3)!1, Parent(x)!3);
> O;
Order of F over Univariate Polynomial Ring in x over GF(5)
> Basis(O);
[
    2,
    2*a,
    2*a^2
]
> P<y> := PolynomialRing(O);
> EO := ext<MaximalOrder(O) | y^2 + O!(2*a)>;
> V := KModule(F, 2);
> M := Module([V | [1, 0], [4, 3], [9, 2]]);
> M;
Module over Maximal Order of F over Univariate Polynomial Ring in x over GF(5)
Ideal of Maximal Order of F over Univariate Polynomial Ring in x over GF(5)
Generator:
1 car Ideal of Maximal Order of F over Univariate Polynomial Ring in x over
GF(5)
Generator:
2
> O2 := Order(EO, M);
> O2;
Order of Algebraic function field defined over F by
$.1^2 + 2*a over Maximal Order of F over Univariate Polynomial Ring in x over
GF(5)
Transformation of EO
Transformation Matrix:
[[ 1, 0, 0 ] [ 0, 0, 0 ]]
[[ 0, 0, 0 ] [ 1, 0, 0 ]]
```

```
> Basis(O2);
[ 1, $.1 ]
```

42.2.3 Orders and Ideals

Orders may be created using ideals of other orders. Ideals are discussed in Section 42.12.

`MultiplicatorRing(I)`

Returns the multiplicator ring of the ideal I of the order O, that is, the subring of elements of the field of fractions of O that multiply I into itself.

`pMaximalOrder(O, p)`

The p-maximal over order of O where p is a prime polynomial or ideal of the coefficient ring of O or an element of valuation 1 of the valuation ring.

If O is a Kummer extension then specific code is used to calculate each p-maximal order, rather than the Round 2 method. In this case we know 1 or 2 elements which generate the p-maximal order and can write the order down.

If O is an Artin–Schreier extension then we can also write down a basis for the p-maximal order and avoid the Round 2 algorithm. We use [Sti93] Proposition III.7.8 to get a start on computing these elements.

`pRadical(O, p)`

Returns the p-radical of an order O for a prime p (polynomial or ideal of the coefficient ring or element of valuation 1 of the valuation ring), defined as the ideal consisting of elements of O for which some power lies in the ideal pO.

It is possible to call this function even if p is not prime. In this case the p-trace-radical will be computed, i.e.

$$\{x \in F \mid \mathrm{Tr}(xO) \subseteq C\}$$

for F the field of fractions of O and C the order of p (if p is an ideal) or the parent of p otherwise. If p is square free and all divisors are larger than the field degree, this is the intersection of the radicals for all l dividing p.

42.3 Related Structures

42.3.1 Parent and Category

Function fields form the Magma category `FldFun` and function field orders form the Magma category `RngFunOrd`. The notional power structures exist as parents of function fields and their orders but allow no operations.

`Category(F)` `Category(O)`

`Parent(F)` `Parent(O)`

42.3.2 Other Related Structures

More interesting related structures (than above) are listed below.

`PrimeRing(F)`

`PrimeField(F)`

`PrimeRing(O)`

The prime field of the function field F or the order O (prime ring of the constant field).

`ConstantField(F)`

`DefiningConstantField(F)`

The constant field k, where $F = k(x, \alpha)$.

`ExactConstantField(F)`

The exact constant field of the algebraic function field F/k, i.e. the algebraic closure in F of the constant field k of F, together with the inclusion map.

`BaseRing(F)`

`BaseField(F)`

`CoefficientRing(F)`

`CoefficientField(F)`

The rational function field $k(x)$ if the function field F is an extension of $k(x)$ and k if F is an extension of k. If F is an extension of another algebraic function field then this field will be returned.

`ISABaseField(F,G)`

Applies to more general fields within Magma than function fields. Returns whether G is amongst the recursively defined base fields of F.

`BaseRing(O)`

`CoefficientRing(O)`

The polynomial algebra $k[x]$ if the order O is finite or the degree valuation ring if O is infinite. If O is an extension of another order of an algebraic function field this order will be returned.

`BaseRing(FF)`

`BaseField(FF)`

`CoefficientRing(FF)`

`CoefficientField(FF)`

Given a field of fractions FF of an order O return the field of fractions of the coefficient ring of O.

`SubOrder(O)`

For a non equation order O returns the order which O was created as a transformation of. This order is one transformation closer to the equation order.

`FunctionField(O)`

The function field which O is an order of.

`FieldOfFractions(O)`

`FieldOfFractions(FF)`

`FieldOfFractions(F)`

Given an order O, this function returns the field of fractions, a field with the same basis as O. On a function field or a field of fractions this function is trivial.

`Order(FF)`

Given a field of fractions FF return the order O which is the ring of integers of FF.

`RationalExtensionRepresentation(F)`

The function field F represented as an extension of a rational function field. This function gives the representation of function fields F/k as finite extensions.

`AbsoluteOrder(O)`

The order O as an extension of its bottom coefficient ring, (i.e. the order of the `RationalExtensionRepresentation` of the field of fractions of O corresponding to O).

`AbsoluteFunctionField(F)`

The function field F expressed as an extension of its constant field.

`UnderlyingRing(F)`

`UnderlyingField(F)`

`UnderlyingRing(F, R)`

`UnderlyingField(F, R)`

Return the underlying ring of the function field F over R. This is F expressed as an extension of R. If R is not given then it is taken to be the coefficient field of the coefficient field of F. The field R must appear in the tower of coefficient fields under F.

`Embed(F, L, a)`

`Embed(F, L, s)`

Install the embedding of F into L with the image(s) of the primitive element(s) of F being the element a in L or the images in s in L.

`Places(F)`

The set of places of the algebraic function field F/k.

`DivisorGroup(F)`

The group of divisors of the algebraic function field F/k.

`DifferentialSpace(F)`

The space of differentials of the algebraic function field F/k.

Example H42E8

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> f := y^3 + (4*x^3 + 4*x^2 + 2*x + 2)*y^2 + (3*x + 3)*y + 2;
> F<alpha> := FunctionField(f);
> ConstantField(F);
Finite field of size 5
> CoefficientField(F);
Univariate rational function field over GF(5)
Variables: x
> CoefficientRing(MaximalOrderFinite(F));
Univariate Polynomial Ring in x over GF(5)
> FieldOfFractions(IntegralClosure(ValuationRing(R), F));
Algebraic function field defined over Univariate rational function field over
GF(5)
Variables: x by
y^3 + (4*x^3 + 4*x^2 + 2*x + 2)*y^2 + (3*x + 3)*y + 2
> Order(IntegralClosure(ValuationRing(R), F),
>     MatrixAlgebra(CoefficientRing(MaximalOrderInfinite(F)), 3)!4,
>     CoefficientRing(MaximalOrderInfinite(F))!1);
```

```
Maximal Order of F over Valuation ring of Univariate rational function field
over GF(5) with generator 1/x
> SubOrder($1);
Maximal Order of F over Valuation ring of Univariate rational function field
over GF(5) with generator 1/x
> $1 eq $2;
false
> Places(F);
Set of places of F
> DivisorGroup(F);
Divisor group of F
```

Example H42E9

Output from `UnderlyingRing` is shown.

```
> PF<x> := PolynomialRing(GF(31, 3));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x^3 + 1>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b*x*y - 1>;
> RationalExtensionRepresentation(FF2);
Algebraic function field defined over Univariate rational function field over
GF(31^3) by
y^6 + 29*y^3 + (30*x^5 + x^2)*y^2 + 1
> UnderlyingRing(FF2);
Algebraic function field defined over Univariate rational function field over
GF(31^3) by
y^6 + 29*y^3 + (30*x^5 + x^2)*y^2 + 1
> UnderlyingRing(FF2, FieldOfFractions(PF));
Algebraic function field defined over GF(31^3) by
$.1^6 + 29*$.1^3 + 30*$.1^2*$.2^5 + $.1^2*$.2^2 + 1
```

`WeilRestriction(E, n)`

`Reduction`	BoolElt	*Default :* `true`
Verbose	`WeilRes`	*Maximum :* 1

A hyperelliptic function field in the Weil restriction over $\mathbf{F}_q$ of the elliptic function field E: $y^2+xy+x^3+ax^2+b$ defined over $\mathbf{F}_{q^n}$ where q is a power of 2. Also returns a function which can be used to map a place (not a pole or zero of x) of F into a divisor of the result. See [Gau00]. `Reduction` indicates whether a (possibly quite expensive) reduction step is performed at the end of the computation. It defaults to `true`.

`ConstantFieldExtension(F, E)`

Return the function field with constant field E which contains the function field F. The ring E must cover the constant field of F. If E is contained in the exact constant field of F then F and the new field will be isomorphic.

Example H42E10

Changing the constant field to the exact constant field is shown below.

```
> P<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(y^6 + y + 2);
> E<a> := ExactConstantField(F);
> C, r := ConstantFieldExtension(F, E);
> r(c);
1/16*(a^5 + 4*a^4 + 6*a^3 + 4*a^2 + a)
> $1 @@ r;
c
> e := Random(C, 2);
> e @@ r;
1/2*x*c^5 - 3*x*c^4 + (-12*x + 8)*c^3 + (-16*x + 24)*c^2 + (-8*x + 16)*c - 1
> r($1);
1/2*(-a^5 - a^2 + a)*$.1 + a^5 + a^4 - a^3 - a^2 - 1
```

`Reduce(O)`

Given an order O belonging to a function field F, this function returns the order obtained by applying size-reduction to the basis of O.

42.4 General Structure Invariants

`Characteristic(F)`

`Characteristic(O)`

The characteristic of the function field F/k or one of its orders O.

`IsPerfect(F)`

Applies to any field in Magma. Returns whether F is perfect.

`Degree(F)`

`Degree(F, G)`

`Degree(O)`

The degree $[F : G]$ of the field extension F/G where G is the base field of F unless specified. For an order O, this function returns the rank of O as a module over its coefficient ring. Note that this rank is equal to the degree $[F : G]$ where F and G are the field of fractions of O and the coefficient ring of O respectively.

`AbsoluteDegree(F)`

`AbsoluteDegree(O)`

The degree of the function field F or the order O as a finite extension of $k(x)$ or $k[x]$ or as an infinite extension of k.

`DefiningPolynomial(F)`

`DefiningPolynomial(O)`

The defining polynomial of the function field F over its coefficient ring. For an order O belonging to a function field F, this function returns the defining polynomial of O, which may be different from that of $F/k(x, \alpha_1, \ldots, \alpha_r)$.

`DefiningPolynomials(F)`

`DefiningPolynomials(O)`

Return the defining polynomials of the function field F or the order O as a sequence of polynomials over the coefficient ring.

`Basis(F)`

`Basis(O)`

`Basis(O, R)`

The basis $1, \alpha, \ldots, \alpha^{n-1}$ of the function field $F[\alpha]$ over the coefficient field.

Given an order O belonging to a function field F, this function returns the basis of O in the form of function field elements.

Given an additional ring R, return the basis of O as elements of R.

`TransformationMatrix(O1, O2)`

Return the matrix M and a denominator d which transforms elements of the order $O1$ into elements of the order $O2$.

`CoefficientIdeals(O)`

The coefficient ideals of the order O of a relative extension. These are the ideals $\{A_i\}$ of the coefficient ring of O such that for every element e of O, $e = \sum_i a_i * b_i$ where $\{b_i\}$ is the basis returned for O and each $a_i \in A_i$.

`BasisMatrix(O)`

Given an order O in a function field F of degree n, this returns an $n \times n$ matrix whose i-th row contains the coefficients for the i-th basis element of O with respect to the power basis of F. Thus, if b_i is the i-th basis element of O,

$$b_i = \sum_{j=1}^{n} M_{ij} \alpha^{j-1}$$

where M is the matrix and α is the generator of F.

`PrimitiveElement(O)`

A root of the defining polynomial of the order O.

`Discriminant(O)`

The discriminant of the order O, up to a unit in its coefficient ring.

`AbsoluteDiscriminant(O)`

The discriminant of the order O of an algebraic function field F over the bottom coefficient ring of O, (the subring of the rational function field F extends).

`DimensionOfExactConstantField(F)`
`DegreeOfExactConstantField(F)`

The dimension of the exact constant field of the function field F/k over k. The exact constant field is the algebraic closure of k in F.

`Genus(F)`

The genus of the function field F/k.

Example H42E11

```
> PF<x> := PolynomialRing(GF(31, 3));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x^3 + 1>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b*x*y - 1>;
> Characteristic(FF2);
31
> EFF2I := EquationOrderInfinite(FF2);
> MFF2I := MaximalOrderInfinite(FF2);
> Degree(MFF2I) eq 3;
true
> AbsoluteDegree(EFF2I);
6
> Genus(FF2);
9
> DefiningPolynomial(EFF2I);
$.1^3 + [ 0, 30/x^3 ]*$.1 + [ 30/x^9, 0 ]
> Basis(MFF2I);
[ 1, 1/x*d, 1/x^2*d^2 ]
> Discriminant(EFF2I);
Ideal of Maximal Equation Order of FF1 over Valuation ring of Univariate
rational function field over GF(31^3)
Variables: x with generator 1/x
Generator:
(4*x^3 + 27)/x^15*b + 4/x^18
> AbsoluteOrder(EFF2I);
```

```
Order of Algebraic function field defined over Univariate rational function
field over GF(31^3) by
y^6 + 29*y^3 + (30*x^5 + x^2)*y^2 + 1 over Valuation ring of Univariate rational
function field over GF(31^3) with generator 1/x
> AbsoluteDiscriminant(EFF2I);
(2*x^9 + 25*x^6 + 6*x^3 + 29)/x^33
> Discriminant($2);
(30*x^24 + 6*x^21 + 16*x^18 + 20*x^15 + 16*x^12 + 7*x^9 + 27*x^6 + 3*x^3 +
    30)/x^48
```

Example H42E12

Invariants are slightly different for non–simple fields.

```
> P<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(P);
> F<a, b> := FunctionField([3*y^3 - x^2, x*y^2 + 1]);
> DefiningPolynomials(F);
[
    3*y^3 - x^2,
    x*y^2 + 1
]
> DefiningPolynomials(EquationOrderFinite(F));
[
    y^3 - 1/3*x^2,
    y^2 + x
]
> DefiningPolynomials(EquationOrderInfinite(F));
[
    $.1^3 - 1/3/$.1^4,
    $.1^2 + 1/$.1
]
> Basis(F);
[
    1,
    a,
    a^2,
    $.1*b,
    $.1*a*b,
    $.1*a^2*b
]
> TransformationMatrix(EquationOrderFinite(F), MaximalOrderFinite(F));
[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 x 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 x 0]
[0 0 0 0 0 x]
```

```
1
> TransformationMatrix(MaximalOrderFinite(F), EquationOrderFinite(F));
[x 0 0 0 0 0]
[0 x 0 0 0 0]
[0 0 1 0 0 0]
[0 0 0 x 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
x
```

`GapNumbers(F)`

`SeparatingElement` FldFunGElt *Default :*

The sequence of global gap numbers of the function field F/k (in characteristic zero this is always $[1, \ldots, g]$). A separating element used internally for the computation can be specified, it defaults to `SeparatingElement(F)`. See the description of `GapNumbers` on page 1165.

`GapNumbers(F, P)`

The sequence of gap numbers of the function field F/k at P where P must be a place of degree one. See the description of `GapNumbers` on page 1165.

`SeparatingElement(F)`

Returns a separating element of the function field F/k.

`RamificationDivisor(F)`

`SeparatingElement` FldFunGElt *Default :*

The ramification divisor of the function field F/k. The semantics of calling `RamificationDivisor()` with F or the zero divisor of F are identical. For further details see the description of `RamificationDivisor` on page 1167.

`WeierstrassPlaces(F)`

`SeparatingElement` FldFunGElt *Default :*

The Weierstrass places of the function field F/k. The semantics of calling `WeierstrassPlaces` with F or the zero divisor of F are identical. See the description of `WeierstrassPlaces` on page 1168.

`WronskianOrders(F)`

`SeparatingElement` FldFunGElt *Default :*

The Wronskian orders of the function field F/k. The semantics of calling `WronskianOrders` with F or the zero divisor of F are identical. See the description of `WronskianOrders` on page 1168.

```
Different(O)
```

The different of the maximal order O.

```
Index(O, S)
```

The index of S in O where S is a suborder of O and O and S have the same equation order.

42.5 Galois Groups

Finding Galois groups (of normal closures) of polynomials over rational function fields over $k \in \{\mathbf{Q}, \mathbf{F}_q\}$, where $\mathbf{F}_q$ denotes the finite field of characteristic p with $q = p^r$, $r \in \mathbf{Z}_{>0}$ is a hard problem, in general. All practical algorithms used the classification of transitive groups, which is known up to degree 31 [CHM98]. These algorithms fall into two groups: The absolute resolvent method [SM85] and the method of Stauduhar [Sta73].

The Magma implementation is based on an extension of the method of Stauduhar by Klüners, Geißler [Gei03, GK00] and, more recently, Fieker [FK] and Sutherland [Sut]. There is no longer any limit on the degree of the polynomials or fields as this algorithm does not use the classification of transitive groups. In contrast to the absolute resolvent method, it also provides the explicit action on the roots of the polynomial f which generates the function field. The algorithm strongly depends on the fact that the corresponding problem is implemented for the residue class field.

Roughly speaking, the method of Stauduhar traverses the subgroup lattice of transitive permutation groups of degree n from the symmetric group to the actual Galois group. This is done by using so-called relative resolvents. Resolvents are polynomials whose splitting fields are subfields of the splitting field of the given polynomial which are computed using approximations of the roots of the polynomial f.

If the field (or the field defined by a polynomial) has subfields (i.e. the Galois group is imprimitive) the current implementation changes the starting point of the algorithm in the subgroup lattice, to get as close as possible to the actual Galois group. This is done via computation of subfields of a stem field of f, that is the field extension of $k(t)$ which we get by adjoining a root of f to $k(t)$. The Galois group is found as a subgroup of the intersection of suitable wreath products (using the knowledge of the subfields) which may be easily computed.

If the field (or the field defined by a polynomial) does not have subfields (i.e. the Galois group is primitive) we use a combination of the method of Stauduhar and the absolute resolvent method. The Frobenius automorphism of the underlying field already determines a subgroup of the Galois group, which is used to speed up computations in the primitive case.

The algorithms used here are similar to those use for number fields. See also Chapter 38. In addition to the intrinsics described here, some of the intrinsics described in Section 38.2 apply to polynomials over function fields also.

`GaloisGroup(f)`

`Prime`	RngElt	*Default :*
`NextPrime`	UserProgram	*Default :*
`ProofEffort`	RngIntElt	*Default :* 10
`Ring`	GaloisData	*Default :*
`ShortOK`	BoolElt	*Default :* `false`
`Old`	BoolElt	*Default :* `false`
Verbose	`GaloisGroup`	*Maximum :* 5

Given a separable, (irreducible if k is $\mathbf{Q}$) polynomial $f(t,x)$ of degree n over the rational function field $k(t)$, $k \in \{\mathbf{Q}, \mathbf{F}_q\}$, or an extension K thereof (if $k = \mathbf{F}_q$) this function returns a permutation group that forms the Galois group of a splitting field for f in some algebraic closure of K. The permutation group acts on the points $1, 2, \ldots, n$. The roots of f are calculated in the process, expressed as power series and returned as the second argument: For a prime polynomial $p(t) \in k(t)$ denote by $\bar{N}$ the splitting field of the polynomial $f(t,x) \bmod p(t)$. It is well known that the roots of the polynomial $f(t,x)$ can be expressed as power series in $\bar{N}[[t]]$. We embed $\bar{N}$ in an unramified p-adic extension. The third return value is a structure containing information about the computation that can be used to compute the roots of f to arbitrary precision. This can be used for example in `GaloisSubgroup` on page 977 to compute arbitrary subfields of the splitting field.

The required precision increases linearly with the index of the subgroups, which are passed traversing the subgroup lattice. Therefore computations may slow down considerably for higher degrees and large indices.

The default version employs series computations over either unramified p-adic fields ($k = \mathbf{Q}$) or finite fields ($k = \mathbf{F}_q$). The prime polynomial is determined during a Galois group computation in such a way that f is squarefree modulo p.

The prime to use for splitting field computations can be given via the parameter `Prime`. The method of choosing primes for splitting field computations can be given by the parameter `NextPrime`. An indication of how much effort the computation should make to prove the results can be provided by altering the parameter `ProofEffort`, it should be increased if more effort should be made.

If `Old` is set to `true`, then the old version is called if available. Since the return values of the new version differ substantially from the old one, this may be used in old applications.

`GaloisGroup(F)`

`Prime`	RngElt	*Default :*
`NextPrime`	UserProgram	*Default :*
`ProofEffort`	RngIntElt	*Default :* 10
`Ring`	GaloisData	*Default :*
`ShortOK`	BoolElt	*Default :* `false`

Given a function field F defined as an extension of either a rational function field or a global algebraic function field by one polynomial compute the Galois group of a normal closure of F.

The prime to use for splitting field computations can be given via the parameter `Prime`. The method of choosing primes for splitting field computations can be given by the parameter `NextPrime`. An indication of how much effort the computation should make to prove the results can be provided by altering the parameter `ProofEffort`, it should be increased if more effort should be made.

Example H42E13

A Galois group computation is shown below.

```
> k<t>:= FunctionField(Rationals());
> R<x>:= PolynomialRing(k);
> f:= x^15 + (-1875*t^2 - 125)*x^3 + (4500*t^2 + 300)*x^2 +
>     (-3600*t^2 - 240)*x + 960*t^2+ 64;
> G, r, S:= GaloisGroup(f);
> TransitiveGroupDescription(G);
1/2[S(5)^3]S(3)
> A := Universe(r);
> AssignNames(~A,  ["t"]);
> A;
Power series ring in t over Unramified extension
defined by the polynomial (1 + O(191^20))*x^4 +
    O(191^20)*x^3 + (7 + O(191^20))*x^2 + (100 +
    O(191^20))*x + 19 + O(191^20)
 over Unramified extension defined by the
polynomial (1 + O(191^20))*x + 190 + O(191^20)
 over pAdicField(191)
> r[1];
> r[1];
-54*$.1^3 + 68*$.1^2 + 31*$.1 - 12 + O(191) +
    (-15*$.1^3 - 66*$.1^2 - 2*$.1 - 39 + O(191))*t
    + O(t^2)
> S;
GaloisData of type p-Adic (FldFun over Q)
> TransitiveGroupIdentification(G);
99 15
```

Example H42E14

Some examples for polynomials over rational function fields over finite fields

```
> k<x>:= FunctionField(GF(1009));
> R<y>:= PolynomialRing(k);
> f:= y^10 + (989*x^4 + 20*x^3 + 989*x^2 + 20*x + 989)*y^8 + (70*x^8 +
> 869*x^7 + 310*x^6 + 529*x^5 + 600*x^4 + 479*x^3 + 460*x^2 + 719*x +
> 120)*y^6 + (909*x^12 + 300*x^11 + 409*x^10 + 1000*x^9 + 393*x^8 +
> 657*x^7 + 895*x^6 + 764*x^5 + 420*x^4 + 973*x^3 + 177*x^2 + 166*x +
> 784)*y^4 + (65*x^16 + 749*x^15 + 350*x^14 + 909*x^13 + 484*x^12 +
> 452*x^11 + 115*x^10 + 923*x^9 + 541*x^8 + 272*x^7 + 637*x^6 + 314*x^5 +
> 724*x^4 + 490*x^3 + 948*x^2 + 99*x + 90)*y^2 + 993*x^20 + 80*x^19 +
> 969*x^18 + 569*x^17 + 895*x^16 + 101*x^15 + 742*x^14 + 587*x^13 +
> 55*x^12+ 437*x^11 + 97*x^10 + 976*x^9 + 62*x^8 + 171*x^7 + 930*x^6 +
> 604*x^5 + 698*x^4 + 60*x^3 + 60*x^2 + 1004*x + 1008;
> G, r, p:= GaloisGroup(f);
> t1, t2:= TransitiveGroupIdentification(G);
> t1;
1
> t2;
10
```

And a second one.

```
> k<t>:= FunctionField(GF(7));
> R<x>:= PolynomialRing(k);
> f:= x^12 + x^10 + x^8 + (6*t^2 + 3)*x^6 + (4*t^4 + 6*t^2 + 1)*x^4 +
> (5*t^4 + t^2)*x^2 + 2*t^4;
> G, r, p:= GaloisGroup(f);
> G;
Permutation group G acting on a set of cardinality 12
    (2, 8)(3, 9)(4, 10)(5, 11)
    (1, 5, 9)(2, 6, 10)(3, 7, 11)(4, 8, 12)
    (1, 12)(2, 3)(4, 5)(6, 7)(8, 9)(10, 11)
> A := Universe(r);
> AssignNames(~A,  ["t"]);
> r;
[
    w^950*t^13 + w^1350*t^12 + w^1900*t^11 + w^500*t^10 + w^2050*t^9 + 2*t^8 +
        w^1350*t^7 + w^300*t^6 + w^350*t^5 + w^1450*t^4 + w^950*t^3 + w^1000*t^2
        + w^1100*t + w^550,
    w^1175*t^13 + w^1825*t^12 + w^1675*t^11 + w^725*t^10 + w^1025*t^9 +
        w^1825*t^8 + w^1325*t^7 + w^775*t^6 + w^1775*t^5 + w^1325*t^4 +
        w^1575*t^3 + w^1175*t^2 + w^2225*t + w^2275,
    w^25*t^13 + w^1075*t^12 + w^425*t^11 + w^925*t^10 + w^225*t^9 + w^2375*t^8 +
        w^2125*t^7 + w^625*t^6 + w^1175*t^5 + w^425*t^4 + w^575*t^3 + w^825*t^2
        + w^1175*t + w^2375,
    w^175*t^13 + w^1525*t^12 + w^575*t^11 + w^475*t^10 + w^1575*t^9 + w^1025*t^8
        + w^475*t^7 + w^775*t^6 + w^1025*t^5 + w^1775*t^4 + w^1625*t^3 +
        w^2175*t^2 + w^1025*t + w^1025,
    w^1025*t^13 + w^1975*t^12 + w^2125*t^11 + w^1475*t^10 + w^2375*t^9 +
        w^1975*t^8 + w^2075*t^7 + w^1825*t^6 + w^425*t^5 + w^875*t^4 +
        w^1425*t^3 + w^2225*t^2 + w^1175*t + w^325,
    w^650*t^13 + w^2250*t^12 + w^100*t^11 + w^1100*t^10 + w^1150*t^9 + 2*t^8 +
        w^1050*t^7 + w^2100*t^6 + w^1250*t^5 + w^550*t^4 + w^650*t^3 +
        w^2200*t^2 + w^1700*t + w^1450,
    w^2150*t^13 + w^150*t^12 + w^700*t^11 + w^1700*t^10 + w^850*t^9 + 5*t^8 +
        w^150*t^7 + w^1500*t^6 + w^1550*t^5 + w^250*t^4 + w^2150*t^3 +
        w^2200*t^2 + w^2300*t + w^1750,
    w^2375*t^13 + w^625*t^12 + w^475*t^11 + w^1925*t^10 + w^2225*t^9 + w^625*t^8
        + w^125*t^7 + w^1975*t^6 + w^575*t^5 + w^125*t^4 + w^375*t^3 +
        w^2375*t^2 + w^1025*t + w^1075,
    w^1225*t^13 + w^2275*t^12 + w^1625*t^11 + w^2125*t^10 + w^1425*t^9 +
        w^1175*t^8 + w^925*t^7 + w^1825*t^6 + w^2375*t^5 + w^1625*t^4 +
        w^1775*t^3 + w^2025*t^2 + w^2375*t + w^1175,
    w^1375*t^13 + w^325*t^12 + w^1775*t^11 + w^1675*t^10 + w^375*t^9 +
        w^2225*t^8 + w^1675*t^7 + w^1975*t^6 + w^2225*t^5 + w^575*t^4 +
        w^425*t^3 + w^975*t^2 + w^2225*t + w^2225,
    w^2225*t^13 + w^775*t^12 + w^925*t^11 + w^275*t^10 + w^1175*t^9 + w^775*t^8
        + w^875*t^7 + w^625*t^6 + w^1625*t^5 + w^2075*t^4 + w^225*t^3 +
        w^1025*t^2 + w^2375*t + w^1525,
    w^1850*t^13 + w^1050*t^12 + w^1300*t^11 + w^2300*t^10 + w^2350*t^9 + 5*t^8 +
        w^2250*t^7 + w^900*t^6 + w^50*t^5 + w^1750*t^4 + w^1850*t^3 + w^1000*t^2
        + w^500*t + w^250
]
> p;
t^2 + 4
```

42.6 Subfields

For finite extensions L of $\mathbf{Q}(t)$, $\mathbf{F}_q(t)$ or extensions K of $\mathbf{F}_q(t)$ Magma can compute all fields between L and $\mathbf{Q}(t)$, $\mathbf{F}_q(t)$ or K. It should be noted that the computation of subfields does not depend on the Galois groups. The implementation over $\mathbf{Q}(t)$ uses the algorithm of Klüners [Klü02]. The implementation over $\mathbf{F}_q(t)$ and extensions thereof follows the newer ideas of Klüners and van Hoeij [vHKN11].

```
Subfields(F)
```

All algebraic function fields G with $k(x) \subset G \subseteq F$ or $K \subset G \subseteq F$ where K is an extension of $\mathbf{F}_q(x)$ and the coefficient field of F.

Example H42E15

A subfield computation is shown below.

```
> k<x>:= FunctionField(Rationals());
> R<y>:= PolynomialRing(k);
> f:= y^14 - 3234*y^12 + (8*x + 123480)*y^11 + (-696*x - 1152480)*y^10 +
> (27672*x - 43563744)*y^9 + (-663544*x + 1795525424)*y^8 + (10660416*x -
>  33905500608)*y^7 + (-120467088*x + 409661347536)*y^6 + (976911040*x -
>  3428257977088)*y^5 + (-5684130144*x + 20264929189344)*y^4 + (23251514496*x -
>  83582683562112)*y^3 + (-63672983360*x + 229899367865216)*y^2 +
>  (105037027200*x - 380160309247488)*y - 79060128000*x + 286518963720192;
> F:= FunctionField(f);
> Subfields(F);
[
    <Algebraic function field defined over Univariate rational function field
    over Rational Field
    Variables: x by
    y^14 - 3234*y^12 + (8*x + 123480)*y^11 + (-696*x - 1152480)*y^10 + (27672*x
        - 43563744)*y^9 + (-663544*x + 1795525424)*y^8 + (10660416*x -
        33905500608)*y^7 + (-120467088*x + 409661347536)*y^6 + (976911040*x -
        3428257977088)*y^5 + (-5684130144*x + 20264929189344)*y^4 +
        (23251514496*x - 83582683562112)*y^3 + (-63672983360*x +
        229899367865216)*y^2 + (105037027200*x - 380160309247488)*y -
        79060128000*x + 286518963720192, Mapping from: FldFun: F to FldFun: F>,
    <Algebraic function field defined over Univariate rational function field
    over Rational Field
    Variables: x by
    y^7 + 294*y^6 - 107016*y^5 + (2744*x + 576240)*y^4 + (-806736*x +
        2469418896)*y^3 + (88740960*x - 312072913824)*y^2 + (-4329483200*x +
        15606890921216)*y + 79060128000*x - 286518963720192, Mapping from:
    Algebraic function field defined over Univariate rational function field
    over Rational Field
    Variables: x by
    y^7 + 294*y^6 - 107016*y^5 + (2744*x + 576240)*y^4 + (-806736*x +
        2469418896)*y^3 + (88740960*x - 312072913824)*y^2 + (-4329483200*x +
        15606890921216)*y + 79060128000*x - 286518963720192 to FldFun: F>
]
```

42.7 Automorphism Group

Let K be a finite extension of the rational function field over a finite field, the rationals or a number field. In contrast to the number field situation, there are two different natural notions of automorphisms here: we distinguish between automorphisms that fix the base field and arbitrary automorphisms that can also induce non-trivial maps of the constant field.

The first case, automorphisms fixing the base field of K, is analogous to the number field case and was implemented by Jürgen Klüners.

The second case of more general automorphisms has been implemented by Florian Heß along the lines of his paper [Heß04]. Here the constant field of K can, in fact, be any exact perfect field in MAGMA with a few provisos.

42.7.1 Automorphisms over the Base Field

`Automorphisms(K, k)`

Computes all $\mathbf{Q}(t)$ automorphisms of the absolute finite extension K that fix k. The field k has to be $\mathbf{Q}(t)$ for this function.

`AutomorphismGroup(K, k)`

Return the group of k-automorphisms of the algebraic function field K together with the map from the group to the sequence of automorphisms of K. The field k has to be $\mathbf{Q}(t)$.

Example H42E16

We define an extension of degree 7 over $\mathbf{Q}(t)$ and compute the automorphisms.

```
> Q:=Rationals();
> Qt<t>:=PolynomialRing(Q);
> Qtx<x>:=PolynomialRing(Qt);
> f := x^7 + (t^3 + 2*t^2 - t + 13)*x^6 + (3*t^5 - 3*t^4
>     + 9*t^3 + 24*t^2 - 21*t + 54)*x^5 + (3*t^7 -
>     9*t^6 + 27*t^5 - 22*t^4 + 6*t^3 + 84*t^2 -
>     121*t + 75)*x^4 + (t^9 - 6*t^8 + 22*t^7 -
>     57*t^6 + 82*t^5 - 70*t^4 - 87*t^3 + 140*t^2 -
>     225*t - 2)*x^3 + (-t^10 + 5*t^9 - 25*t^8 +
>     61*t^7 - 126*t^6 + 117*t^5 - 58*t^4 - 155*t^3
>     + 168*t^2 - 80*t - 44)*x^2 + (-t^10 + 8*t^9 -
>     30*t^8 + 75*t^7 - 102*t^6 + 89*t^5 + 34*t^4 -
>     56*t^3 + 113*t^2 + 42*t - 17)*x + t^9 - 7*t^8
>     + 23*t^7 - 42*t^6 + 28*t^5 + 19*t^4 - 60*t^3 -
>     2*t^2 + 16*t - 1;
> K:=FunctionField(f);
> A:=Automorphisms(K, BaseField(K));
> #A;
7
```

Now we transform this list into a group to see that it is really cyclic. We pass in special functions for equality testing and multiplication to speed the algorithm up.

```
> G := GenericGroup(A: Eq := func<a,b | a`Images eq b`Images>,
>                   Mult := func<a,b | hom<K -> K | a`Images @ b>>);
> G;
Finitely presented group G on 2 generators
Relations
    G.1 = Id(G)
    G.1 * G.2 = G.2 * G.1
    G.1 * G.2^2 = G.2^2 * G.1
    G.1 * G.2^3 = G.2^3 * G.1
    G.1 * G.2^4 = G.2^4 * G.1
    G.1 * G.2^5 = G.2^5 * G.1
    G.1 * G.2^6 = G.2^6 * G.1
    G.1 = G.2^7
```

Finally, we verify that this gives the same result as `AutomorphismGroup`.

```
> AutomorphismGroup(K, BaseField(K));
Finitely presented group on 2 generators
Relations
    $.1 = Id($)
    $.1 * $.2 = $.2 * $.1
    $.1 * $.2^2 = $.2^2 * $.1
    $.1 * $.2^3 = $.2^3 * $.1
    $.1 * $.2^4 = $.2^4 * $.1
    $.1 * $.2^5 = $.2^5 * $.1
    $.1 * $.2^6 = $.2^6 * $.1
    $.1 = $.2^7
Mapping from: GrpFP to [
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K
] given by a rule
```

`IsSubfield(K, L)`

Given two absolute finite extensions K and L of $\mathbf{Q}(t)$, decide if L is an extension of K. If this is the case, return an embedding map from K into L.

`IsIsomorphicOverQt(K, L)`

Given two absolute finite extensions K and L of $\mathbf{Q}(t)$, decide if L is $\mathbf{Q}(t)$-isomorphic to K. If this is the case, return a map from K onto L.

Example H42E17

`Subfields` and `IsIsomorphic` are illustrated below.

```
> Q:=Rationals();
> Qt<t>:=PolynomialRing(Q);
> Qtx<x>:=PolynomialRing(Qt);
> K:=FunctionField(x^4-t^3);
> L:=Subfields(K);
> #L;
2
> L:=L[2][1]; L;
Algebraic function field defined over Univariate
```

```
rational function field over Rational Field
Variables: t by
x^2 - t^3
```

Now we will check if L is indeed a subfield of K:

```
> IsSubfield(L,K);
true Mapping from: FldFun: L to FldFun: K
```

Obviously, L can be defined using a simpler polynomial:

```
> LL:=FunctionField(x^2-t);
> IsIsomorphicOverQt(LL,L);
true Mapping from: FldFun: LL to FldFun: L
```

42.7.2 General Automorphisms

`Isomorphisms(K, E)`

`BaseMorphism`	Map	*Default :* `false`
`Bound`	RngIntElt	*Default :* ∞
`Strategy`	MonStgElt	*Default :* "*None*"

Given two function fields K and E, this function computes a list of at most `Bound` field isomorphisms from K to E.

If `BaseMorphism` is given it should be an isomorphism f between the constant fields of K and E. In this case only isomorphisms extending f are considered.

The default behaviour is for all isomorphisms from K to E which extend SOME isomorphism of the constant field of K to that of E considered. In this case (no base morphism is specified), the constant fields must be finite, the rationals or a number field. If the base morphism f is specified then the constant fields can be any exact perfect fields (finite or characteristic 0).

If the base morphism f is specified, it can be defined in the natural way for most constant field types. For example, for finite fields and number fields, the usual `hom<k->l|x>`, where `x` gives the image of $k.1$, can be used. A common situation is where the constant fields of K and E are equal to k and f is the identity. This can be defined very simply for any k by `IdentityFieldMorphism(k)` . Several more intrinsics related to field morphisms are described in the following subsection.

The possible choices of `Strategy` are `"None"`, `"Weierstrass"` or `"DegOne"`. If `Strategy` is different to `"None"`, this determines the places that are used as the basis of the construction of the maps. In all cases, a finite set of places of E and K which must correspond under any isomorphism are used. All isomorphisms are found between the canonical affine models (as defined by Heß) obtained by omitting one of these places from each of E and K.

`DegOne` can only be used with finite constant fields. In this case, isomorphisms are determined which map a fixed degree one place of K to any one of the finite number of degree one places of E. This function can fail in rare situations if the constant field of K is too small and no degree one place exists. In this case an appropriate error message is displayed.

`Weierstrass` uses the Weierstrass places of the fields. Isomorphisms are determined which map a fixed Weierstrass place of K to any of those of E with the same degree and Riemann-Roch data. This strategy can be very fast if the residue field and Riemann-Roch data of a particular place of K match those of only a few (or no!) Weierstrass places of E.

In case of fields of genus < 2, the constant field must be finite.

`IsIsomorphic(K, E)`

`BaseMorphism`	Map	*Default :* `false`
`Strategy`	MonStgElt	*Default :* "*None*"

As above, except the function only computes a single isomorphism if one exists.

`Automorphisms(K)`

`BaseMorphism`	Map	*Default :* `false`
`Bound`	RngIntElt	*Default :* ∞
`Strategy`	MonStgElt	*Default :* "*None*"

This function computes a list of at most `Bound` automorphisms of the function field K. This is an abbreviation for `Isomorphisms(K, K)` and the parameters are as described above.

An important difference is that the BaseMorphism, if specified, must be of field morphism type. `IdentityFieldMorphism` may be used, but basic constrictors for non-trivial constant field maps f will usually cause an error if used directly. The way around this is to use the conversion `f := FieldMorphism(f)` (see the following subsection).

`Isomorphisms(K,E,p1,p2)`

`Automorphisms(K,p1,p2)`

`Bound`	RngIntElt	*Default :* ∞

As above except that the constant field morphism is taken as the identity and only iso/automorphisms which take function field place $p1$ to $p2$ are computed.

`AutomorphismGroup(K)`

`BaseMorphism`	Map	*Default :* `false`
`Strategy`	MonStgElt	*Default :* "*None*"

Given a function field K, this function computes that group of automorphisms satisfying the conditions specified by the parameters and returns it as a finitely-presented group. The map also returned is invertible and takes a group element to the function field isomorphism that it represents.

`AutomorphismGroup(K,f)`

Strategy MonStgElt *Default : "None"*

In this variation, the automorphism group of the function field K is computed in its permutation representation on a set of places or divisors or in its linear representation on a space of differentials or subspace of K.

The return values consist of the representing group G, a map (with inverse) from G to the maps of K giving the actual isomorphisms, and a sequence of isomorphisms of K which consist of the kernel of the representation.

Only automorphisms fixing the constant field are considered here. If the set/space on which the representation is to be defined is not invariant by the automorphism group, a run-time error will result.

The argument$f$ should be a map defining the representation.

Its domain must be an enumerated sequence for a permutation representation or a vector space for a linear representation.

Its codomain should be K or a space or enumerated sequence of elements of K, places of K, divisors of K or differentials of K. The examples below show some common ways of producing f by using functions like `DifferentialSpace` and `RiemannRochSpace`.

42.7.3 Field Morphisms

The isomorphisms returned by the functions in the last subsection are of general `Map` type but contain some extra internal structure. The same is true of the maps used to specify `BaseMorphism`. These objects come in two flavours: *field morphisms*, that represent maps between general fields, and the more specialised *function field morphisms*, representing maps between algebraic function fields. This subsection contains several related functions that are very useful when working with (function) field morphisms.

`IsMorphism(f)`

Returns `true`, if the map is a field or function field morphism; `false` otherwise.

`FieldMorphism(f)`

Converts a homomorphism between fields into a field morphism.

`IdentityFieldMorphism(F)`

Returns the identity automorphism of field F as a field morphism.

`IsIdentity(f)`

Returns `true` if f is the identity morphism; `false` otherwise.

`Equality(f, g)`

Returns `true`, if the two maps are both field morphisms or function field morphisms and are equal; `false` otherwise.

`HasInverse(f)`

Either returns "`true`" and the inverse morphism for (function) field morphism f, or "`false`" if inverse does not exist, or "`unknown`" if it cannot be computed.

`Composition(f, g)`

The composition of the field morphisms f and g.

Example H42E18

We illustrate the use of the general isomorphism functions with some examples. In the first, we have a rational function field of characteristic 5:

```
> k<w> := GF(5);
> kxf<x> := RationalFunctionField(k);
> kxfy<y> := PolynomialRing(kxf);
> F<a> := FunctionField(x^2+y^2-1);
> L := Isomorphisms(kxf, F);
> #L eq #PGL(2, k);
```

In the next example we consider the function field of a hyperelliptic curve defined over $\mathbf{Q}(i)$ $[i^2 = -1]$ and a Galois twist of it. The fields are not isomorphic over $\mathbf{Q}(i)$ but they are over $\mathbf{Q}$:

```
> k<i> := QuadraticField(-1);
> kxf<x> := RationalFunctionField(k);
> kxfy<y> := PolynomialRing(kxf);
> F1<a> := FunctionField(y^2-x^5-x^2-i);
> F2<b> := FunctionField(i*y^2-x^5-i*x^2+1);
> c := IdentityFieldMorphism(k);
> IsIsomorphic(F1,F2 : BaseMorphism := c);
false
> IsIsomorphic(F1,F2);
true Mapping from: FldFun: F1 to FldFun: F2 given by a rule
> L := Isomorphisms(F1, F2);
> [<f(a), f(x), f(i)> : f in L];
[
    <-i*b, i*x, -i>,
    <i*b, i*x, -i>
]
```

In the next example we consider the function field of the genus 3 plane curve $x^3*y+y^3*z+z^3*x = 0$, which has full automorphism group $PGL_2(\mathbf{F}_7)$. We compute automorphisms over different finite fields and also compute the automorphisms group as an FP group.

```
> k := GF(11);
> kxf<x> := RationalFunctionField(k);
> kxfy<y> := PolynomialRing(kxf);
> K<y> := FunctionField(x^3*y+y^3+x);
> L := Automorphisms(K);
> #L;
3
```

```
> // Extend base field to get all autos
> k := GF(11^3);
> kxf<x> := RationalFunctionField(k);
> kxfy<y> := PolynomialRing(kxf);
> K<y> := FunctionField(x^3*y+y^3+x);
> L := Automorphisms(K);
> #L;
504
> // restrict to just "geometric" autos, which fix the base
> c := IdentityFieldMorphism(k);
> L := Automorphisms(K : BaseMorphism := c);
> #L;
168
> // get the automorphism group instead as an FP group
> G,mp := AutomorphismGroup(K : BaseMorphism := c);
> G;
Finitely presented group G on 2 generators
Relations
    G.2^3 = Id(G)
    (G.1^-1 * G.2)^3 = Id(G)
    G.1^7 = Id(G)
    (G.2^-1 * G.1^-3)^2 = Id(G)
    (G.2^-1 * G.1^-1)^4 = Id(G)
> #G;
168
> IdentifyGroup(G); // find in small group database
<168, 42>
```

Finally, we give an example of a genus 1 function field over $\mathbf{F}_5$ where the group of automorphisms is computed acting on various spaces of functions and differentials.

```
> k<w> := GF(5);
> kxf<x> := RationalFunctionField(k);
> kxfy<y> := PolynomialRing(kxf);
> f := x^3 + y^3 + 1;
> F<a> := FunctionField(f);
> f := Numeration(Set(Places(F, 1)));
> G, h, K := AutomorphismGroup(F, f);
> #G; Type(G);
12
GrpPerm
> V, f := SpaceOfDifferentialsFirstKind(F);
> G, h, K := AutomorphismGroup(F, f);
> #G; Type(G);
2
GrpMat
> D := &+ Places(F, 1);
> V, f := DifferentialSpace( -D );
> G, h := AutomorphismGroup(F, f);
> #G;
12
> V, f := RiemannRochSpace( D );
> G, h, ker := AutomorphismGroup(F, f);
> #G; #ker;
12
1
```

42.8 Global Function Fields

F/k denotes a global function field in this section.

42.8.1 Functions relative to the Exact Constant Field

`NumberOfPlacesOfDegreeOverExactConstantField(F, m)`

`NumberOfPlacesDegECF(F, m)`

The number of places of degree m of the global function field F/k. Contrary to the `Degree` function the degree is here taken over the respective exact constant fields.

`NumberOfPlacesOfDegreeOneOverExactConstantField(F)`

`NumberOfPlacesOfDegreeOneECF(F)`

The number of places of degree one in the global function field F/k. Contrary to the `Degree()` function the degree is here taken over the exact constant field.

`NumberOfPlacesOfDegreeOneOverExactConstantField(F, m)`

`NumberOfPlacesOfDegreeOneECF(F, m)`

The number of places of degree one in the constant field extension of degree m of the global function field F/k. Contrary to the `Degree()` function the degree is here taken over the respective exact constant fields.

`SerreBound(F)`

`SerreBound(F, m)`

`SerreBound(q, g)`

The Serre bound on the number of places of degree one in a global function field of genus g over the exact constant field of q elements (of the global function field F, of the constant field extension of degree m of F). Contrary to the `Degree()` function the degree is here taken over the respective exact constant fields.

```
IharaBound(F)
IharaBound(F, m)
IharaBound(q, g)
```

The Ihara bound on the number of places of degree one in a global function field F/k of genus g over the exact constant field of q elements (of the global function field F, of the constant field extension of degree m of F). Contrary to the `Degree` function the degree is here taken over the respective exact constant fields.

```
NumberOfPlacesOfDegreeOneECFBound(F)
NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(F)
NumberOfPlacesOfDegreeOneECFBound(F, m)
NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(F, m)
NumberOfPlacesOfDegreeOneECFBound(q, g)
NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(q, g)
```

The minimum of the Serre and Ihara bound. Contrary to the `Degree` function the degree is here taken over the respective exact constant fields.

```
LPolynomial(F)
```

The L-polynomial of the global function field F/k (with respect to the exact constant field).

```
LPolynomial(F, m)
```

The L-polynomial of the constant field extension of degree m of the global function field F/k (with respect to the exact constant field).

```
ZetaFunction(F)
```

The Zeta function of the global function field F/k (with respect to the exact constant field).

```
ZetaFunction(F, m)
```

The Zeta function of the constant field extension of degree m of the global function field F/k (with respect to the exact constant field).

42.8.2 Functions Relative to the Constant Field

`Places(F, m)`

A sequence containing the places of degree m of the global function field F/k.

`HasPlace(F, m)`

Returns `true` and a place of degree m if and only if there exists such a place in the global function field; `false` otherwise.

`HasRandomPlace(F, m)`

Returns `true` and a random place of degree m in the global function field (`false` if there are none).

`RandomPlace(F, m)`

Returns a random place of degree m in the global function field or throws an error if there is none.

Example H42E19

```
> Y<t> := PolynomialRing(Integers());
> R<x> := FunctionField(GF(9));
> P<y> := PolynomialRing(R);
> f := y^3 + y + x^5 + x + 1;
> F<alpha> := FunctionField(f);
> Genus(F);
4
> NumberOfPlacesDegECF(F, 1);
22
> NumberOfPlacesOfDegreeOneECFBound(F);
32
> HasRandomPlace(F, 2);
true (x^2 + $.1*x + 2, alpha + $.1^2*x + $.1^5)
> LPolynomial(F);
6561*t^8 + 8748*t^7 + 7290*t^6 + 3888*t^5 + 1539*t^4 + 432*t^3 + 90*t^2
    + 12*t + 1
```

Example H42E20

Some of the above functions are demonstrated for a global relative field.

```
> PF<x> := PolynomialRing(GF(13, 2));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b>;
> RER_FF2 := RationalExtensionRepresentation(FF2);
> NumberOfPlacesOfDegreeOneECF(FF2) eq NumberOfPlacesOfDegreeOneECF(RER_FF2);
```

```
true
> SerreBound(FF2);
170
> NumberOfPlacesDegECF(FF2, 1);
170
> _, P := HasPlace(FF2, 1);
> P;
(x, (($.1^44*x + $.1^100)*b + ($.1^82*x + $.1^10))*d^2 + (($.1^85*x + $.1^67)*b
    + ($.1^107*x + $.1^130))*d + ($.1^26*x + $.1^69)*b + $.1^149*x)
> Degree(P) eq 1;
true
> LPolynomial(FF2, 2) eq LPolynomial(RER_FF2, 2);
true
```

42.8.3 Functions related to Class Group

`UnitRank(O)`

Given a maximal ‘finite’ order O in a global function field, return the unit rank of O.

`UnitGroup(O)`

The unit group of a ‘finite’ maximal order O as an Abelian group and the map from the unit group into O. Also see `IsUnitWithPreimage` on page 1131.

`Regulator(O)`

The regulator of the unit group of the ‘finite’ maximal order O.

`PrincipalIdealMap(O)`

The map from the multiplicative group of the field of fractions of O to the group of fractional ideals of O where O is a ‘finite’ maximal order.

Example H42E21

Following on from the last example,

```
> EFF2F := EquationOrderFinite(FF2);
> G, m := UnitGroup(EFF2F);
> G;
Abelian Group isomorphic to Z/168
Defined on 1 generator
Relations:
    168*G.1 = 0
> m(Random(G));
[ [ $.1^120, 0 ], [ 0, 0 ], [ 0, 0 ] ]
> IsUnit($1);
true
```

```
> Regulator(EFF2F);
1
```

`ClassGroup(F :parameters)`

DegreeBound	RngIntElt	*Default :*
SizeBound	RngIntElt	*Default :*
ReductionDivisor	DivFunElt	*Default :*
Proof	BoolElt	*Default :*

The divisor class group of F/k as an Abelian group, a map of representatives from the class group to the divisor group and the homomorphism from the divisor group onto the divisor class group. For a detailed description see `ClassGroup` on page 1170.

`ClassGroup(O)`

The ideal class group of the 'finite' maximal order O as an Abelian group, a map of representatives from the ideal class group to the group of fractional ideals and the homomorphism from the group of fractional ideals onto the ideal class group.

`ClassGroupExactSequence(O)`

Returns the maps in the center of the exact sequence

$$0 \to U \to F^\times \to Id \to Cl \to 0$$

where U is the unit group of O, $F^\times$ is the multiplicative group of the field of fractions of O, Id is the group of fractional ideals of O and Cl is the class group of O for a 'finite' maximal order O.

`ClassGroupAbelianInvariants(F :parameters)`

DegreeBound	RngIntElt	*Default :*
SizeBound	RngIntElt	*Default :*
ReductionDivisor	DivFunElt	*Default :*
Proof	BoolElt	*Default :*

Computes a sequence of integers containing the Abelian invariants of the divisor class group of F/k. For a detailed description see `ClassGroupAbelianInvariants` on page 1170.

`ClassGroupAbelianInvariants(O)`

Computes a sequence of integers containing the Abelian invariants of the ideal class group of the 'finite' maximal order O.

`ClassNumber(F)`

The order of the group of divisor classes of degree zero of F/k.

`ClassNumber(O)`

The order of the ideal class group of the 'finite' maximal order O.

Example H42E22

An example of class groups of relative fields is shown.

```
> PF<x> := PolynomialRing(GF(13, 2));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b : Check := false>;
> MFF2I := MaximalOrderInfinite(FF2);
> G, m, mi := ClassGroup(FF2);
> m(Random(G));
Divisor in reduced representation:
Divisor in ideal representation:
Fractional ideal of Maximal Equation Order of FF2 over Maximal Equation Order of
FF1 over Univariate Polynomial Ring in x over GF(13^2)
Generators:
1
($.1^60/x*b + $.1^57/x)*d^2 + ($.1^141/x*b + $.1^4/x)*d + $.1^80/x*b, Ideal of
MFF2I
Generators:
1
1,
-2,
6*(1/x, (($.1^132*x^2 + $.1^164*x + 12)/x^3*b + ($.1^85*x^2 + $.1^155*x +
    12)/x^3)*d^2 + (($.1^75*x^2 + $.1^81*x + 12)/x^3*b + ($.1^29*x^2 + $.1^155*x
    + 12)/x^3)*d + ($.1^163*x^2 + $.1^29*x + 12)/x^3*b + ($.1^141*x + 12)/x^2),
    (x)^2 * (1/x)^2
> mi(&+[Divisor(Random(FF2, 3)) : i in [1 .. 3]]);
0
> ClassNumber(FF2);
1
```

`GlobalUnitGroup(F)`

The group of global units of F/k, i. e. the multiplicative group of the exact constant field, as an Abelian group, together with the map into F. Also see `IsGlobalUnit` on page 1131 and `IsGlobalUnitWithPreimage` on page 1131.

`ClassGroupPRank(F)`

Compute the p-rank of the class group of F/k where p is the characteristic of F/k. For a detailed description see `ClassGroupPRank` on page 1173.

`HasseWittInvariant(F)`

Return the Hasse–Witt invariant of F/k. See `HasseWittInvariant` on page 1173 for a detailed description.

`IndependentUnits(O)`

A sequence of independent units of the 'finite' maximal order O.

`FundamentalUnits(O)`

A sequence of fundamental units of the 'finite' maximal order O.

Example H42E23

```
> R<x> := FunctionField(GF(3));
> P<y> := PolynomialRing(R);
> f := y^4 + x*y + x^4 + x + 1;
> F<a> := FunctionField(f);
> O := MaximalOrderFinite(F);
> Basis(O);
[ 1, a, a^2, a^3 ]
> Discriminant(O);
x^12 + x^3 + 1
> UnitRank(O);
1
> U := FundamentalUnits(O);
> U;
[ [ x^33 + x^31 + 2*x^30 + 2*x^28 + 2*x^27 + x^25 + 2*x^24 + x^22 + 2*x^19 +
    2*x^15 + x^10 + 2*x^9 + 2*x^7 + x^6 + 2*x + 2, x^32 + 2*x^30 + x^29 + 2*x^28
    + 2*x^27 + 2*x^26 + x^22 + x^21 + 2*x^19 + x^18 + x^17 + x^16 + x^13 + x^11
    + 2*x^10 + 2*x^9 + 2*x^3 + 1, x^29 + x^27 + 2*x^25 + 2*x^23 + x^22 + 2*x^21
    + x^20 + x^18 + 2*x^17 + x^16 + x^15 + 2*x^14 + x^11 + 2*x^10 + 2*x^4 + x,
    x^30 + 2*x^27 + x^24 + x^21 + 2*x^18 + x^9 + 2*x^6 + 2 ] ]
> Norm(U[1]);
1
> Regulator(O);
33
```

42.9 Structure Predicates

`IsField(R)` `IsEuclideanDomain(R)`

`IsPID(R)` `IsUFD(R)`

`IsDivisionRing(R)` `IsEuclideanRing(R)`

`IsPrincipalIdealRing(R)` `IsDomain(R)`

`F eq G` `F ne G` `O1 eq O2` `O1 ne O2`

`O1 subset O2`

Return whether $O1$ is a subset of $O2$.

`IsGlobal(F)`

Returns `true` if and only if the algebraic function field F/k is global, i.e. the constant field is a finite field; `false` otherwise.

`IsRationalFunctionField(F)`

Return `true` if the function field F is isomorphic to a rational function field, (i.e. F is only trivially algebraic).

`IsFiniteOrder(O)`

Given an order O of a function field, return `true` if and only if the bottom coefficient ring of O is a polynomial ring.

`IsEquationOrder(O)`

Given an order O of a function field, return `true` if and only if the order O is an equation order (i.e. it has been defined by a polynomial and so has a power basis).

`IsAbsoluteOrder(O)`

Return `false` if the order O is an extension of another order, otherwise `true`.

`IsMaximal(O)`

Given an order O of a function field, return `true` if and only if the order O is maximal in its field of fractions.

`IsTamelyRamified(O)`

Return whether the order O is tamely ramified, i.e. no prime ideal of O has residue field with characteristic dividing its ramification index.

`IsTotallyRamified(O)`

Return whether there is an ideal of the order O which is totally ramified, i.e. its ramification index is equal to the degree of O over its coefficient ring.

```
IsUnramified(O)
```

Return whether a finite order O is unramified at the finite places and whether an infinite order O is unramified at the infinite places.

```
IsWildlyRamified(O)
```

Return whether there is a prime ideal of the order O which is wildly ramified, i.e. its ramification index is divisible by the characteristic of its residue class field.

```
IsInKummerRepresentation(K)
```

Tests if the global function field K is, in its current representation, a Kummer extension. More specific, this function tests if the defining polynomial is of the form $x^r - a$ for some r coprime to the characteristic and if r divides the order of the multipicative group of the constant field, ie. if the coefficient ring of K contains a primitive r-th root of unity. In case K is in Kummer representation, the element a is returned as a second return value.

```
IsInArtinSchreierRepresentation(K)
```

Tests if a global function field K is, in its current representation, a Artin-Schreier extension, ie. if the defining polynomial of K is of the form $x^p - x - a$ where p is the characteristic of K. In this case, the element a is returned as a second return value.

42.10 Homomorphisms

```
hom< F -> R | g >
hom< F -> R | cf, g >
```

The homomorphism from the function field F to any ring R where g is the image of the generator of F in R and cf is a map from the coefficient field of F into R.

```
hom<  O -> R | g  >
hom<  O -> R | cf, g  >
```

Create the map from the order O of an algebraic function field to R using g as the image of the primitive element of O. If the map cf is given it should be from the coefficient ring of O into R, otherwise the coefficient ring of O should be automatically coercible into R.

```
IsRingHomomorphism(m)
```

Return whether the vector space homomorphism m is a homomorphism of rings.

Example H42E24

A simple use of homomorphisms is shown.

```
> PR<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(PR);
> FR1<a> := FunctionField(y^3 - x*y + 1);
> P<y> := PolynomialRing(FR1);
> FR2<c> := FunctionField(y^2 - a^5*x^3*y + 1);
> EFR2F := EquationOrderFinite(FR2);
> cf := hom<FR1 -> EFR2F | a + 1>;
> h := hom<FR2 -> EFR2F | cf, c + 1>;
> h(c) eq c + 1;
true
> h(a*c) eq a*c + a + c + 1;
true
```

hom< O -> R | $b_1, ..., b_n$ >

hom< O -> R | m, $b_1, ..., b_n$ >

Return the map from the order O of an algebraic function field into the ring R which maps the basis elements of O to $b_1, ..., b_n$. The map m, if given, should be from the coefficient ring of O into R and will be used to map the coefficients of the basis elements. If not given, the coefficient ring of O should automatically coerce into R.

42.11 Elements

The function field $F = k(x, \alpha_1, \ldots, \alpha_r, \alpha)$ may be viewed as n-dimensional vector space over $k(x, \alpha_1, \ldots, \alpha_r)$, where n is the degree of the field extension $F/k(x, \alpha_1, \ldots, \alpha_r)$. Note that F is spanned by the powers $1, \alpha, \ldots, \alpha^{n-1}$. Within Magma, function field elements are printed as linear combinations of these powers of α over the coefficient field.

An order can be viewed as a free R-module of rank n where R is its coefficient ring (a polynomial ring or the degree valuation ring of $k(x)$ or an order which is a lesser degree extension of $k[x]$) and n equals the degree $F/k(x, \alpha_1, \ldots, \alpha_r)$. It has a basis consisting of n elements. Within Magma, function field order elements are printed as a sequence of coefficients of the R-linear combination of such a basis.

Elements can also be represented as a product of other function field or order elements. This is referred to as the product representation. Product representations can be useful for large elements, however, it is expensive to put such elements in a set or to test them for equality as this involves finding coefficients for the element.

42.11.1 Creation of Elements

```
F . 1
F . 2
```

(i) Return the generator for the function field F over $k(x, \alpha_1, \ldots, \alpha_r)$, that is, $\alpha \in F$ such that $F = k(x, \alpha_1, \ldots, \alpha_r, \alpha)$.

(ii) Return the first and second generators for the function field F over k, that is, $\alpha \in F$ and $x \in F$ such that $F = k(x, \alpha)$.

```
Name(F, i)
```

Given a function field F, return the i-th generator, i.e. return the element `F.1` or `F.2` of F.

```
O . i
FF . i
```

Return the ith basis element of the order O or its field of fractions FF.

```
F ! a
elt< F | a >
```

Create the element of the function field F specified by a; here a is allowed to be an element coercible into F, which means that a may be:

(i) an element of F,

(ii) an element of the coefficient field of F,

(iii) an element of another representation of F.

For F an extension of $k(x)$ we additionally have

(iv) an element of an order of F,

(v) an element being coercible into $k(x)$,

(vi) a sequence of elements being coercible into the coefficient field of F of length equal to the degree of F over its coefficient field. In this case the element $a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1}$ is created, where $a = [a_0, \ldots, a_{n-1}]$ and α is the generator `F.1` of F over its coefficient field.

```
O ! a
elt< O | a >
```

Create the element of the order O specified by a; here a is allowed to be an element coercible into O, which means that mathematically $a \in O$ and that a may be any of:

(i) an element of the function field F,

(ii) an element of an order of the function field F,

(iii) an element that can be coerced into $k(x)$,

(iv) an element that can be coerced into its coefficient field,

(v) a sequence of elements being coercible into the coefficient field of O of length equal to the rank of O over its coefficient field. In this case the element $a_1\omega_1 + a_2\omega_2 + \ldots + a_n\omega_n$ is created, where $a = [a_1, \ldots, a_n]$ and $\omega_1, \omega_2, \ldots, \omega_n$ is the basis of O as returned by `Basis(O)`.

```
FF ! a
elt< FF | a >
```

Create the element of the field of fractions FF of an order O specified by a, where a may be any of the above such that $d * a$ is mathematically in O for some $d \in O$.

`elt< F | a`$_0$`, a`$_1$`, ..., a`$_{n-1}$` >`

Create the element $a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1}$ where $a_0, \ldots, a_{n-1}$ are coercible into the coefficient field of the function field F, n equals the degree of F over its coefficient field and α is the generator `F.1` of F over $k(x)$.

`elt< O | a`$_1$`, a`$_2$`, ..., a`$_n$` >`

`elt< FF | a`$_1$`, a`$_2$`, ..., a`$_n$` >`

Create the element $a_1\omega_1 + a_2\omega_2 + \ldots + a_n\omega_n$ where $a_1, \ldots, a_n$ are coercible into the coefficient ring of the order O, n equals the rank of O over its coefficient ring and $\omega_1, \omega_2, \ldots, \omega_n$ is the basis of O as returned by `Basis(O)`, (where O is the ring of integers of the field of fractions FF).

```
One(F)              One(O)
Identity(F)         Identity(O)
Zero(F)             Zero(O)
Representative(F)   Representative(O)
```

These generic functions (cf. Chapter 17) create 1, 1, 0, and 0 respectively in the function field F or order O.

```
Random(F, m)
Random(O, m)
```

A “random” element of the global function field F or one of its orders O. The size of the coefficients of the element are determined by m.

42.11.2 Parent and Category

```
Parent(a)           Category(a)
```

42.11.3 Sequence Conversions

The sequence conversions refer to the function field F as a vector space of dimension n over the coefficient field of F where F is a finite degree extension (degree n) of its coefficient field.

`ElementToSequence(a)`

`Eltseq(a)`

The sequence $[a_1, \ldots, a_n]$ of elements of the coefficient field of the parent of the function field or order element a such that $a = a_1\omega_1 + a_2\omega_2 + \ldots + a_n\omega_n$ where $\omega_1, \omega_2, \ldots, \omega_n$ is a basis of the parent of a.

`Eltseq(a, R)`

A sequence of coefficients of the function field element a in the coefficient field R.

`Flat(a)`

A sequence of coefficients of the function field element a in the bottom coefficient field of the parent of a.

`F ! [ a`$_0$`, a`$_1$`, ..., a`$_{n-1}$` ]`

The element $a = a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1}$ where the function field $F = k(x, \alpha_1, \ldots, \alpha_r, \alpha)$ and the a_i may be coerced into $k(x, \alpha_1, \ldots, \alpha_r)$.

`O ! [ a`$_1$`, a`$_2$`, ..., a`$_n$` ]`

The element $a = a_1\omega_1 + a_2\omega_2 + \ldots + a_n\omega_n$ where $\omega_1, \omega_2, \ldots, \omega_n$ is a basis of the order O and the a_i are coercible into the coefficient ring of O.

Example H42E25

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> f := y^3 + (4*x^3 + 4*x^2 + 2*x + 2)*y^2 + (3*x + 3)*y + 2;
> F<alpha> := FunctionField(f);
> Evaluate(f, alpha);
0
> F.1;
alpha
> b := x + alpha + 1/x*alpha^2;
> b;
1/x*alpha^2 + alpha + x
> b eq F ! [x, 1, 1/x];
true
```

42.11.4 Arithmetic Operators

The following binary arithmetic operations can also be performed in the case where one operand is an element of the function field F or an order O and the other operand is a ring element which can naturally be mapped into F or O.

`+ a` `- a`

`a + b` `a - b` `a * b` `a div b` `a / b` `a ^ k`

`Modexp(a, k, m)`

Return a^k *mod* m where m is an element of $k[x]$ or o_∞ according to whether the parent of a is a finite or infinite order.

`a mod I`

Return the element a belonging to the order O as an element of O/I.

`Modinv(a, m)`

Return the inverse of the element a of an order of a function field modulo m where m is an element of $k[x]$ or o_∞ according to whether the order of a is a finite or infinite order or an ideal of the order of a.

42.11.5 Equality and Membership

The following binary arithmetic operations can also be performed in the case where one operand is an element of the function field F or an order O and the other operand is a ring element which can naturally be mapped into F or O.

`a eq b` `a ne b`

`a in F` `a in O` `a in FF`

`a notin F` `a notin O` `a notin FF`

42.11.6 Predicates on Elements

The functions in this section list the general ring element predicates that apply to function fields and orders of a function field.

`IsDivisibleBy(a, b)`

Given elements a and b belonging to a function field F or an order O, returns `true` if there exists $c \in F$ or $c \in O$ such that $a = bc$ and returns c as well, provided that $b \neq 0$.

`IsZero(a)` `IsOne(a)` `IsMinusOne(a)`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsUnit(a)` `IsZeroDivisor(a)` `IsRegular(a)`
`IsIrreducible(a)` `IsPrime(a)`

`IsSeparating(a)`

Returns `true` if the function field element a is a separating element (has a non zero differential).

`IsConstant(a)`

Whether the algebraic function a is constant; if so it is returned as an element of the exact constant field.

`IsGlobalUnit(a)`

Whether the function field element a is a global unit, i.e. a constant (equivalent to `IsConstant`)

`IsGlobalUnitWithPreimage(a)`

Returns `true` and the preimage of the function field element a in the global unit group, `false` otherwise. The function field must be global.

`IsUnitWithPreimage(a)`

Returns `true` and the preimage of the order element a in the unit group of O if a is a unit, `false` otherwise. The function field has to be global.

42.11.7 Functions related to Norm and Trace

Multiplication by $a \in F$ or $a \in O$ defines a linear map of the vector space F over its coefficient field where F is a finite extension of its coefficient field. The following functions work with respect to this mapping.

`Trace(a)` `Norm(a)`
`MinimalPolynomial(a)`
`CharacteristicPolynomial(a)`

`RepresentationMatrix(a)`

Returns the matrix $M \in R^{n\times n}$ such that $a\,(\omega_1, \omega_2, \ldots, \omega_n) = (\omega_1, \omega_2, \ldots, \omega_n)\,M$, where $\omega_1, \omega_2, \ldots, \omega_n$ is a R-basis of the parent of the function field or order element a and R is the coefficient ring of the parent of a.

`Trace(a, R)`

The trace of the function field or order element a over R, a coefficient ring or field of the parent of a.

`Norm(a, R)`

The norm of the order or algebraic function field element a over R, a coefficient ring or field of the parent of a.

`CharacteristicPolynomial(a, R)`

The characteristic polynomial of the order or algebraic function field element a over R, a coefficient ring or field of the parent of a.

`MinimalPolynomial(a, R)`

The minimal polynomial of the order or algebraic function field element a over R, a coefficient ring or field of the parent of a.

`AbsoluteMinimalPolynomial(a)`

The minimal polynomial of the function field element a over the rational function field.

`RepresentationMatrix(a, R)`

Returns the matrix $M \in R^{n\times n}$ such that $a\,(\omega_1, \omega_2, \ldots, \omega_n) = (\omega_1, \omega_2, \ldots, \omega_n)\,M$, where $\omega_1, \omega_2, \ldots, \omega_n$ is a R-basis of the parent of the function field or order element a and R is a coefficient ring of the parent of a.

Example H42E26

```
> P<x> := PolynomialRing(Integers());
> N<n> := NumberField(x^6 - 6);
> P<x> := PolynomialRing(N);
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(y^8 - x^3*N.1^5);
> P<y> := PolynomialRing(F);
> F2<d> := FunctionField(y^3 + N.1*F!x - c);
> d^10;
(c^3 - 3*n*x*c^2 + 3*n^2*x^2*c - n^3*x^3)*d
> Norm(d^10);
-120*n^3*x^3*c^7 + 210*n^4*x^4*c^6 - 252*n^5*x^5*c^5 + 1260*x^6*c^4 -
    720*n*x^7*c^3 + (270*n^2*x^8 + n^5*x^3)*c^2 + (-60*n^3*x^9 - 60*x^4)*c +
    6*n^4*x^10 + 270*n*x^5
> Norm(d^10, CoefficientField(F));
13060694016*n^2*x^80 - 21767823360*n^5*x^75 + 97955205120*n^2*x^70 -
    43535646720*n^5*x^65 + 76187381760*n^2*x^60 - 15237476352*n^5*x^55 +
    12697896960*n^2*x^50 - 1209323520*n^5*x^45 + 453496320*n^2*x^40 -
    16796160*n^5*x^35 + 1679616*n^2*x^30
> Trace(d^10, CoefficientField(F));
0
> Trace(d^10);
0
```

42.11.8 Functions related to Orders and Integrality

`IntegralSplit(a, O)`

Split the element function field or order element a into a numerator and denominator with respect to the order O.

`Numerator(a, O)`

The numerator of the function field element a with respect to the order O.

`Numerator(a)`

Given an element a in a field of fractions of an order O return the numerator of a with respect to O.

`Numerator(a, O)`

Given an element a in a field of fractions of an order and an order O of a function field return the numerator of a with respect to O.

`Denominator(a, O)`

The denominator of the function field element a with respect to the order O.

`Denominator(a)`

Given an element a in a field of fractions of an order O return the denominator of a with respect to O.

`Denominator(a, O)`

Given an element a in a field of fractions of an order and an order O of a function field return the denominator of a with respect to O.

`Min(a, O)`

`Minimum(a, O)`

A generator of the ideal $R \cap (d \times a \times O)$ where R is the coefficient ring of the order O and d is the denominator of the function field or order element a wrt O (d is the second return value).

42.11.9 Functions related to Places and Divisors

`Evaluate(a, P)`

Evaluate the algebraic function a at the place P. If it is not defined at P, infinity is returned.

`Lift(a, P)`

Lift the element a of the residue class field of the place P (including infinity) to an algebraic function.

`Valuation(a, P)`

The valuation of the function field or order element a at the place P.

`Expand(a, P)`

`RelPrec`	RngIntElt	*Default :* 10
`AbsPrec`	RngIntElt	*Default :*

Expand the algebraic function a to a series of given precision at the place P and return the local parameter.

`Divisor(a)`
`PrincipalDivisor(a)`

The (principal) divisor (a) of the function field or order element a.

`Zeros(a)`
`Zeroes(a)`

A sequence containing the zeros of the algebraic function a.

`Zeros(F, a)`
`Zeroes(F, a)`

The zeros of the function field element a in the function field F.

`Poles(a)`

A sequence containing the poles of the algebraic function a.

`Poles(F, a)`

A sequence containing the poles of the function field element a in the function field F.

`Degree(a)`

The degree of the algebraic function a, being defined as the degree of the pole (or zero) divisor of a.

`CommonZeros(L)`

Return the common zeros of the function field elements in the sequence L.

`CommonZeros(F, L)`

Return the common zeros in the function field F of the function field elements in the sequence L.

Example H42E27

```
> R<x> := FunctionField(GF(9));
> P<y> := PolynomialRing(R);
> f := y^3 + y + x^5 + x + 1;
> F<a> := FunctionField(f);
> MinimalPolynomial(a);
y^3 + y + x^5 + x + 1
> RepresentationMatrix(a);
[                  0                   1                   0]
[                  0                   0                   1]
[    2*x^5 + 2*x + 2                   2                   0]
> O := IntegralClosure(ValuationRing(R), F);
> Denominator(a, O);
1/x^2
> O := IntegralClosure(PolynomialRing(GF(9)), F);
> Denominator(a, O);
1
> Zeros(a);
[ (x + 2, a), (x^3 + 2*x^2 + 1, a + x^3 + 2*x^2 + 1) ]
> Degree(a);
5
> P := RandomPlace(F, 2);
> P;
(x^2 + $.1^2*x + $.1^6, a + x^2 + $.1^5*x + 1)
> b := Evaluate(a, P);
> b;
$.1^3*$.1 + $.1^3
> c := Lift(b, P);
> c;
$.1^3*x + $.1^3
> Valuation(a, P);
0
> Valuation(a-c, P);
1
```

`Module(L, R)`

IsBasis	BoolElt	*Default :* `false`
PreImages	BoolElt	*Default :* `false`

The R-module generated by the function field elements in the sequence L as an abstract module, together with the map into the algebraic function field. The resulting modules can be used for intersection and inner sum computations.

If the optional parameter `IsBasis` is set `true` the function assumes that the given elements form a basis of the module to be computed.

If the optional parameter `PreImages` is set `true` then the preimages of the given elements under the map are returned as the third return value.

Both optional parameters are mainly used to save computation time.

`Relations(L, R)`

`Relations(L, R, m)`

The module of R-linear relations between the function field elements of the sequence L. The argument m is used for the following: Let the elements of L be $a_1, \ldots, a_n$, V be the relation module $\subseteq R^n$ and define $M := \{ \sum_{i=1}^m v_i a_i \mid v = (v_i)_i \in V \}$. The function tries to compute a generating system of V such that the corresponding generating system of M consists of “small” elements.

`Roots(f, D)`

Compute the roots of the polynomial f which lie in the Riemann-Roch space of the divisor D.

Example H42E28

This example shows the capability of `Module` and `Relations` with relative function fields.

```
> PR<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(PR);
> FR1<a> := FunctionField(y^3 - x);
> P<y> := PolynomialRing(FR1);
> FR2<c> := FunctionField(y^2 - a);
> MFR1F := MaximalOrderFinite(FR1);
> m, f := Module([c, c + a], MFR1F);
> f(m.1);
1
> f(m.2);
c
> m, f := Module([c, c + a], FR1);
> f(m.1);
c
> f(m.2);
1
> m;
KModule m of dimension 2 over FR1
```

```
> Relations([c, c + a], FR1, 1);
Vector space of degree 2, dimension 0 over FR1
User basis:
Matrix with 0 rows and 2 columns
```

42.11.10 Other Operations on Elements

`ProductRepresentation(a)`

Return a product representation for the function field or order element a.

`ProductRepresentation(Q, S)`

`PowerProduct(Q, S)`

Return the element given by the product representation of function field elements in the sequence Q and exponents in the sequence S. It is expensive to put large elements in product representation into sets or to test for them for equality.

`RationalFunction(a)`

`RationalFunction(a, R)`

Return the algebraic function a as a rational function in free variables with respect to the defining polynomial over the coefficient field.

If the ring R is provided it must appear in the tower of coefficient fields of the parent of a and the result is a polynomial over R with respect to all the defining polynomials of the extensions in between.

`Differentiation(x, a)`

The first differentiation resp. derivative of the function field element a with respect to the separating element x.

`Differentiation(x, n, a)`

The nth differentiation of the function field element a with respect to the separating element x. In characteristic zero the nth differentiation equals the nth derivative times $1/n!$.

`DifferentiationSequence(x, n, a)`

The 0-th up to the n-th differentiation of the function field element a with respect to the separating element x.

```
PrimePowerRepresentation(x, k, a)
```

Return the coefficients of the representation of the function field element a as a linear combination of k-th prime powers and powers of the function field element x. More precisely, let $p > 0$ be the characteristic of F. Then F^{p^k} is a subfield of F of index p^k and F can be viewed as a F^{p^k}-vector space. A basis is given by $1, x, \ldots, x^{p^k-1}$ for x a separating element. The function returns $\lambda_1, \ldots, \lambda_{p^k-1} \in F^{p^k}$ such that $a = \sum_i \lambda_i x^i$.

```
Different(a)
```

The different of the element a of an order of an algebraic function field.

```
RationalReconstruction(e, f)
```

For an element e of some some function field K with integral coefficients $e = \sum e_i \alpha^i$, $e_i \in k[x]$ and some polynomial $f \in k[x]$ find the (essentially) unique $E = \sum E_i \alpha_i$ with $E_i \in k(x)$ and $E_i = e_i \bmod f$, where the numerator and denominator of the E_i have degree bounded by half the degree of f. If such E_i exists they are unique, the corresponding element $\sum E_i \alpha^i$ for the function field will be returned as a second return value, the first being `true` to indicate success. If no such element exists, `false` will be returned.

```
CoefficientHeight(a)
```

```
CoefficientHeight(a)
```

The (naive) height of the element as defined as the largest degree of any coefficient or denominator polynomial occurring in the coefficients of a.

```
CoefficientLength(a)
```

```
CoefficientLength(a)
```

The (naive) length or size of the element, defined as the sum of the degrees of all polynomials occurring as coefficients or denominators in the coefficient representation of a.

Example H42E29

```
> F<z> := GF(13, 3);
> PF<x> := PolynomialRing(F);
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - ConstantField(FF1).1>;
> ProductRepresentation([Random(FF2, 2) : i in [1 .. 3]], [2, 3, 2]);
(((z^476*x + z^319)*b + (z^1861*x + z^439))*d^2 + ((z^348*x + z^931)*b +
    (z^328*x + z^2076))*d + (z^152*x + z^1723)*b + z^1044*x + z^1119)^2 *
(((z^1024*x + z^2085)*b + (z^798*x + z^335))*d^2 + ((z^310*x + z^932)*b +
    (z^281*x + z^1393))*d + (z^1844*x + z^66)*b + z^2127*x + z^1788)^3 *
```

```
(((z^1478*x + z^1782)*b + (z^687*x + z^1898))*d^2 + ((z^560*x + z^425)*b +
    (z^2081*x + z^164))*d + (z^60*x + z^890)*b + z^258*x + z^1739)^2
> ProductRepresentation($1);
[
    ((z^476*x + z^319)*b + (z^1861*x + z^439))*d^2 + ((z^348*x + z^931)*b +
        (z^328*x + z^2076))*d + (z^152*x + z^1723)*b + z^1044*x + z^1119,
    ((z^1024*x + z^2085)*b + (z^798*x + z^335))*d^2 + ((z^310*x + z^932)*b +
        (z^281*x + z^1393))*d + (z^1844*x + z^66)*b + z^2127*x + z^1788,
    ((z^1478*x + z^1782)*b + (z^687*x + z^1898))*d^2 + ((z^560*x + z^425)*b +
        (z^2081*x + z^164))*d + (z^60*x + z^890)*b + z^258*x + z^1739
]
[ 2, 3, 2 ]
> r := Random(FF2, 3);
> RationalFunction(r);
((z^1568*x^2 + z^1591*x + z^1260)*b + (z^746*x^2 + z^1405*x + z^1721))*y^2 +
    ((z^990*x^2 + z^689*x + z^470)*b + (z^1324*x^2 + z^195*x + z^1082))*y +
    (z^331*x^2 + z^1995*x + z^1521)*b + z^1323*x^2 + z^852*x + z^2162
> RationalFunction(r, CoefficientField(FF2));
((z^1568*x^2 + z^1591*x + z^1260)*b + (z^746*x^2 + z^1405*x + z^1721))*$.1^2 +
    ((z^990*x^2 + z^689*x + z^470)*b + (z^1324*x^2 + z^195*x + z^1082))*$.1 +
    (z^331*x^2 + z^1995*x + z^1521)*b + z^1323*x^2 + z^852*x + z^2162
> RationalFunction(r, PF);
(z^1568*x^2 + z^1591*x + z^1260)*$.1^2*$.2 + (z^746*x^2 + z^1405*x +
    z^1721)*$.1^2 + (z^990*x^2 + z^689*x + z^470)*$.1*$.2 + (z^1324*x^2 +
    z^195*x + z^1082)*$.1 + (z^331*x^2 + z^1995*x + z^1521)*$.2 + z^1323*x^2 +
    z^852*x + z^2162
> Differentiation(FF2!x, r);
((z^836*x^2 + z^2140*x + z^1077)/x*b + (z^929*x + z^1405))*d^2 + ((z^258*x^2 +
    z^1238*x + z^287)/x*b + (z^1507*x + z^195))*d + (z^1795*x^2 + z^348*x +
    z^1338)/x*b + z^1506*x + z^852
> Differentiation(FF2!b, r);
((z^1112*x + z^1588)*b + (z^1019*x^2 + z^127*x + z^1260))*d^2 + ((z^1690*x +
    z^378)*b + (z^441*x^2 + z^1421*x + z^470))*d + (z^1689*x + z^1035)*b +
    z^1978*x^2 + z^531*x + z^1521
>  Differentiation(FF2!d, r);
>> Differentiation(FF2!d, r);
                  ^

Runtime error in 'Differentiation': First element must be a separating element
> MFR2I := MaximalOrderInfinite(FF2);
> Different(Numerator(r, MFR2I));
[ [ (z^1095*x^5 + z^849*x^4 + z^111*x^3 + z^853*x^2 + z^224*x + z^1340)/x^6,
    (z^666*x^4 + z^1902*x^3 + z^2086*x^2 + z^1119*x + z^1973)/x^5 ], [
    (z^1673*x^5 + z^699*x^4 + z^1465*x^3 + z^1060*x^2 + z^761*x + z^1979)/x^6,
    (z^1034*x^4 + z^1185*x^3 + z^833*x^2 + z^2044*x + z^1701)/x^5 ], [
    (z^516*x^5 + z^1545*x^4 + z^509*x^3 + z^832*x^2 + z^606*x + z^700)/x^6,
    (z^1033*x^4 + z^983*x^3 + z^1375*x^2 + z^89*x + z^271)/x^5 ] ]
```

42.12 Ideals

Ideals for function field orders O are O-modules $I \subseteq F$ for which there is a $d \in F$ such that $dI \subseteq O$ is a non-zero ideal of O, that is they are fractional ideals of O. Over the coefficient ring of O they are also free modules of rank n, where n equals the degree $[F : k(x, \alpha_1, \ldots, \alpha_r)]$.

42.12.1 Creation of Ideals

```
ideal< O | a1, a2, ... , am >
```

Given an order O, as well as elements $a_1, a_2, \ldots, a_m$ coercible into the field of fractions F of O, create the fractional ideal of O generated by these elements.

Note that, contrary to the general case for the constructors, the right hand side elements are not necessarily contained in the left hand side.

```
ideal< O | T, d >
```

The ideal of the order O of an algebraic function field whose basis is the matrix or dedekind module T over the coefficient ring of O divided by the element d of the denominator ring of O.

```
ideal< O | T, S >
ideal< O | T, I1,...,In >
```

The ideal of the order O of an algebraic function field whose basis is the matrix T over the coefficient ring of O along with the coefficient ideals $I_1, ..., I_n$ or those in S.

```
x * O
O * x
```

Create the ideal $x * O$ where x is coercible into the function field of the order O.

```
Ideal(P)
```

Create a prime ideal corresponding to the place P.

```
Ideals(D)
```

Create two ideals of the 'finite' and 'infinite' maximal order respectively corresponding to the divisor D.

```
O !! I
```

Return the ideal I as an ideal of the order O.

42.12.2 Parent and Category

```
Parent(I)    Category(I)
```

42.12.3 Arithmetic Operators

`I + J` `I * J`

`I / J` `I ^ k`

The ideal J is required to be invertible. The ideal I is required to be invertible for negative k.

`c * I` `I * c` `I / c`

`c / I`

The principal ideal generated by the ring element c divided by the ideal I.

`IdealQuotient(I, J)`

`ColonIdeal(I, J)`

The colon ideal $[I : J]$ of elements which multiply all elements of the ideal J into the ideal I.

`ChineseRemainderTheorem(I1, I2, e1, e2)`

`CRT(I1, I2, e1, e2)`

Returns an element e of the order O such that $(e_1 - e)$ is in the ideal I_1 of O and $(e_2 - e)$ is in the ideal I_2.

42.12.4 Roots of Ideals

`IsPower(I, n)`

Return whether the ideal I has an nth root and if so return an nth root.

`Root(I, n)`

Return the nth root of the ideal I.

`IsSquare(I)`

Return whether the ideal I is a square and if so return a square root.

`SquareRoot(I)`

`Sqrt(I)`

Return a square root of the ideal I.

Example H42E30

A simple creation of an ideal and the use of `IsSquare` is shown below.

```
> P<x> := PolynomialRing(GF(79));
> P<y> := PolynomialRing(P);
> Fa<a> := FunctionField(y^2 - x);
> P<y> := PolynomialRing(Fa);
> Fb<b> := FunctionField(y^2 - a);
> P<y> := PolynomialRing(Fb);
> Fc<c> := FunctionField(y^2 + a*b);
> I := a*b*c*MaximalOrderInfinite(Fc);
> IsSquare(I^2);
true Fractional ideal of Maximal Order of Fc over Maximal Order of Fb over
Maximal Equation Order of Fa over Valuation ring of Univariate rational function
field over GF(79) with generator 1/x
Basis:
Pseudo-matrix over Maximal Order of Fb over Maximal Equation Order of Fa over
Valuation ring of Univariate rational function field over GF(79) with generator
1/x
Fractional ideal of Maximal Order of Fb over Maximal Equation Order of Fa over
Valuation ring of Univariate rational function field over GF(79) with generator
1/x
Generators:
1
((78*x^2 + 71*x + 78)/x^2*a + (23*x^2 + 15*x + 78)/x^2)*b + (67*x^2 + 60*x +
    78)/x^2*a + (3*x^2 + 47*x + 78)/x * ( 1 0 )
Fractional ideal of Maximal Order of Fb over Maximal Equation Order of Fa over
Valuation ring of Univariate rational function field over GF(79) with generator
1/x
Generators:
1
((59*x^2 + 44*x + 78)/x^2*a + (4*x^2 + 48*x + 78)/x^2)*b + (29*x^2 + 46*x +
    78)/x^2*a + (71*x + 78)/x * ( 0 1 )
> _, II := $1;
> II eq I;
true
> MaximalOrderFinite(Fc)!!I;
Ideal of Maximal Order of Fc over Maximal Equation Order of Fb over Maximal
Equation Order of Fa over Univariate Polynomial Ring in x over GF(79)
Generators:
a*b*c
a*b*c
```

42.12.5 Equality and Membership

`I eq J` `I ne J` `I in S` `I notin S`

42.12.6 Predicates on Ideals

`IsZero(I)`

Returns `true` if and only if the ideal I is the zero ideal of the order O.

`IsOne(I)`

Returns `true` if and only if the ideal I is the identity ideal of the order O, i.e. $I = O$.

`IsIntegral(I)`

Returns `true` if and only if the ideal I is integral (a true ideal of its order).

`IsPrime(I)`

Returns `true` if and only if the ideal I is prime.

`IsPrincipal(I)`

Returns `true` and a generator if the fractional ideal I is principal, `false` otherwise. The function field has to be global.

42.12.6.1 Predicates on Prime Ideals

`IsInert(P)`

Return `true` if the inertia degree of the prime ideal P is the degree of its order.

`IsInert(P, O)`

Return `true` if there is an inert ideal in the order O above the prime ideal P.

`IsRamified(P)`

Return `true` if the ramification index of the prime ideal P is not 1.

`IsRamified(P, O)`

Return `true` if there is a ramified ideal in the order O above the prime ideal P.

`IsSplit(P)`

Return `true` if the prime ideal P is not the only ideal lying above the prime ideal it lies above.

`IsSplit(P, O)`

Return `true` if there are at least 2 distinct ideals which lie in the order O above the prime ideal P.

```
IsTamelyRamified(P)
```

Return whether the prime ideal P is not wildly ramified.

```
IsTamelyRamified(P, O)
```

Return whether the prime ideal P is not wildly ramified in the order O.

```
IsTotallyRamified(P)
```

Return whether the ramification index of the prime ideal P is the same as the degree of its order over its coefficient order.

```
IsTotallyRamified(P, O)
```

Return whether there are any totally ramified ideals in the order O lying above the prime ideal P.

```
IsTotallySplit(P)
```

Return whether there are as many ideals as the degree of the order of the prime ideal P lying over the prime P lies over.

```
IsTotallySplit(P, O)
```

Return whether there are as many ideals of the order O which lie above the prime ideal P as the degree of O.

```
IsUnramified(P)
```

Return whether the ramification index of the prime ideal P is 1.

```
IsUnramified(P, O)
```

Return whether all the ideals of the order O which lie above the prime ideal P are unramified.

```
IsWildlyRamified(P)
```

Return whether the ramfication index of the prime ideal P is a multiple of the characteristic of the residue field of P.

```
IsWildlyRamified(P, O)
```

Return whether any of the ideals of the order O which lie above the prime ideal P are wildly ramified.

42.12.7 Further Ideal Operations

`I meet J`

The intersection of the ideals I and J.

`Gcd(I, J)`

Given invertible ideals of an order O, returns the greatest common divisor of the ideals I and J.

`Lcm(I, J)`

Given invertible ideals of an order O, returns the least common multiple of the ideals I and J.

`Factorization(I)`

`Factorisation(I)`

Factorization of the ideal I (as sequence of prime ideal, exponent pairs). The order must be maximal.

`Decomposition(O, p)`

A sequence containing all prime ideals of the order O lying above the prime element or ideal p of the coefficient ring of O.

`Decomposition(O)`

A sequence containing all prime ideals of the 'infinite' maximal order O.

`DecompositionType(O, p)`

Sequence of tuples of residue degrees and ramification indices of the prime ideals of the order O lying over p, a prime polynomial or ideal or element of valuation ring of valuation 1.

`DecompositionType(O)`

Sequence of tuples of residue degrees and ramification indices of the prime ideals of the 'infinite' maximal order O.

`MultiplicatorRing(I)`

Returns the multiplicator ring of the ideal I of the order O, that is, the subring of elements of the field of fractions of O that multiply I into itself.

`pMaximalOrder(O, p)`

The p-maximal over order of the order O where p is a prime ideal of the coefficient ring of O. See also the description in Section 42.2.3.

`pRadical(O, p)`

Returns the p-radical of an order O for a prime ideal p of the coefficient ring of O, defined as the ideal consisting of elements of O for which some power lies in the ideal pO.

It is possible to call this function even if p is not prime. In this case the p-trace-radical will be computed, i.e.

$$\{x \in F \mid \mathrm{Tr}(xO) \subseteq C\}$$

for F the field of fractions of O and C the order of p (if p is an ideal) or the parent of p otherwise. If p is square free and all divisors are larger than the field degree, this is the intersection of the radicals for all l dividing p.

`Valuation(a, P)`

`Valuation(I, P)`

The valuation of a or the ideal I at the prime ideal P. The element a must be coercible into the field of fractions of P's order.

`Order(I)`

The order of the ideal I.

`Denominator(I)`

The "smallest" element d of the coefficient ring of the order O of the ideal I such that $dI \subseteq O$.

`Minimum(I)`

A generator m of the ideal $R \cap dI$ where R is the coefficient ring of the ideal's order and d is the denominator of the ideal I (d is the second return value).

`I meet R`

The intersection of the ideal I with a coefficient ring R of its order.

`IntegralSplit(I)`

The integral ideal dI and d, where d is the denominator of the ideal I.

`Norm(I)`

The norm of the ideal I, as element of the coefficient field of the algebraic function field to which I belongs.

`TwoElement(I)`

Given an ideal I with function field F as the function field of its order O, returns two elements $a, b \in F$ such that $I = aO + bO$.

`Generators(I)`

Given a (fractional) ideal I of the order O, return a sequence of elements of the function field F that generate I as an ideal.

`Basis(I)`
`Basis(I, R)`

A basis of the ideal I as a free module over the coefficient ring of its order, coerced into the ring R if given.

`BasisMatrix(I)`

Let $(b_1, \dots, b_n)$ be the basis of the ideal I and let $(\omega_1, \dots, \omega_n)$ be the basis of the order O. A matrix B with coefficients in the rational function field is returned such that $(b_1, \dots, b_n) = (\omega_1, \dots, \omega_n)B^t$.

`TransformationMatrix(I)`

Let $(b_1, \dots, b_n)$ be the basis of the ideal I and let $(\omega_1, \dots, \omega_n)$ be the basis of the order O. A matrix T with coefficients in the coefficient ring of O and a denominator d are returned such that $(b_1, \dots, b_n) = (\omega_1, \dots, \omega_n)T^t/d$.

`CoefficientIdeals(I)`

The coefficient ideals of the ideal I in a relative extension. These are the ideals $\{A_i\}$ of the coefficient ring of the order of I such that for every element $e \in I$, $e = \sum_i a_i * b_i$ where $\{b_i\}$ is the basis returned for I and each $a_i \in A_i$.

`Different(I)`

The different of the (possibly fractional) ideal I of an order of an algebraic function field.

`Codifferent(I)`

The codifferent of the ideal I. This will be the inverse of the different of I if I is an ideal of a maximal order.

`Divisor(I)`

The divisor corresponding to the ideal factorization of the ideal I.

`Divisor(I, J)`

The divisor corresponding to the ideal factorization of the ideals I and J belonging to the ‘finite’ and ‘infinite’ maximal order.

Example H42E31

```
> PR<x> := FunctionField(Rationals());
> P<y> := PolynomialRing(PR);
> FR1<a> := FunctionField(y^3 - x + 1/x^3);
> P<y> := PolynomialRing(FR1);
> FR2<c> := FunctionField(y^2 - a/x^3*y + 1);
> I := ideal<MaximalOrderFinite(FR2) |
> [ x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
> 468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237, x^9 + 1639*x^8 +
> 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 + 468363837*x^3 +
> 242625823*x^2 + 68744019*x + 8052237, (x^15 + 1639*x^14 + 863249*x^13 +
> 148609981*x^12 + 404988066*x^11 + 567876948*x^10 + 468363837*x^9 +
> 242625823*x^8 + 68744019*x^7 + 8052237*x^6)*c, (x^15 + 1639*x^14 +
> 863249*x^13 + 148609981*x^12 + 404988066*x^11 + 567876948*x^10 +
> 468363837*x^9 + 242625823*x^8 + 68744019*x^7 + 8052237*x^6)*c ]>;
> I;
Ideal of Maximal Order of FR2 over Maximal Order of FR1 over Univariate
Polynomial Ring in x over Rational Field
Basis:
Pseudo-matrix over Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x over Rational
Field
Generator:
x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
    468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237 * ( 1 0 )
Fractional ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Generator:
(x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
    468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237)/x^2 * ( 0 1 )
> J := ideal<MaximalOrderFinite(FR2) |
> [ x^3 + 278*x^2 + 164*x + 742, x^3 + 278*x^2 + 164*x + 742, (x^9 + 278*x^8 +
> 164*x^7 + 742*x^6)*c, (x^9 + 278*x^8 + 164*x^7 + 742*x^6)*c ]>;
> J;
Ideal of Maximal Order of FR2 over Maximal Order of FR1 over Univariate
Polynomial Ring in x over Rational Field
Basis:
Pseudo-matrix over Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x over Rational
Field
Generator:
x^3 + 278*x^2 + 164*x + 742 * ( 1 0 )
Fractional ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Generator:
```

```
(x^3 + 278*x^2 + 164*x + 742)/x^2 * ( 0 1 )
> Generators(J);
[
    x^3 + 278*x^2 + 164*x + 742,
    x^3 + 278*x^2 + 164*x + 742,
    (x^7 + 278*x^6 + 164*x^5 + 742*x^4)*c,
    (x^7 + 278*x^6 + 164*x^5 + 742*x^4)*c
]
> TwoElement(J);
x^3 + 278*x^2 + 164*x + 742
((3/2*x^10 + 419*x^9 + 802*x^8 + 1441*x^7 + 1484*x^6)*a^2 + (3/2*x^8 + 417*x^7 +
    246*x^6 + 1113*x^5)*a + (3/2*x^8 + 837/2*x^7 + 663*x^6 + 1359*x^5 +
    1113*x^4))*c + (x^6 + 277*x^5 - 114*x^4 + 578*x^3 - 742*x^2)*a^2 + (3*x^5 +
    834*x^4 + 492*x^3 + 2226*x^2)*a - x^3 - 278*x^2 - 164*x - 742
> Minimum(I);
Ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x over Rational
Field
Generator:
x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
    468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237 1
> Basis(J);
[ 1, x^6*c ]
> Basis(I);
[ 1, x^6*c ]
> I eq J;
false
> II, d := IntegralSplit(I);
> II;
Ideal of Maximal Order of FR2 over Maximal Order of FR1 over Univariate
Polynomial Ring in x over Rational Field
Basis:
Pseudo-matrix over Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x over Rational
Field
Generator:
x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
    468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237 * ( 1 0 )
Fractional ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Generator:
(x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
    468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237)/x^2 * ( 0 1 )
> d;
1
> IsIntegral(I);
true
> GCD(I, J)*LCM(I, J) eq I*J;
```

```
true
```

42.12.7.1 Functions on Prime Ideals

`RamificationIndex(I)`

`RamificationDegree(I)`

The ramification index of the prime ideal I over the corresponding prime of its coefficient ring.

`Degree(I)`

`InertiaDegree(I)`

`ResidueClassDegree(I)`

The residue class degree (inertia degree) of the prime ideal I over the corresponding prime of its coefficient ring.

`ResidueClassField(I)`

The residue class field of the prime ideal I and the residue class mapping.

`Place(I)`

The place corresponding to the prime ideal I, where I is defined over the 'finite' or 'infinite' maximal order.

`SafeUniformizer(P)`

For an ideal I of a maximal order in a function field, this returns an element which has valuation 1 at the given prime and which has valuation 0 at all other primes lying over the same prime of the underlying rational function field. See also `LocalUniformizer` (for places) below.

Example H42E32

```
> R<x> := FunctionField(GF(3));
> P<y> := PolynomialRing(R);
> f := y^4 + x*y + x^4 + x + 1;
> F<a> := FunctionField(f);
> O := MaximalOrderFinite(F);
> x*O;
Ideal of O
Generator:
x
> L := Factorization(x*O);
> L;
[ <Ideal of O
Generators:
```

```
x
a^2 + a + 2, 1>, <Ideal of O
Generators:
x
a^2 + 2*a + 2, 1> ]
> P1 := L[1][1];
> P2 := L[2][1];
> BasisMatrix(P1);
[x 0 0 0]
[0 x 0 0]
[2 1 1 0]
[1 1 0 1]
> P1 meet P2 eq x*O;
true
> IsPrime(P1);
true
> Place(P1);
(x, a^2 + a + 2)
```

42.13 Places

42.13.1 Creation of Structures

`Places(F)`

The set of places of the algebraic function field F/k.

42.13.2 Creation of Elements

42.13.2.1 General Function Field Places

`Decomposition(F, P)`

A sequence containing all places of F/k lying above the place P of the coefficient field $k(x)$ of F. The function field F must be a finite extension of $k(x)$.

`DecompositionType(F, P)`

Sequence of tuples of residue degrees and ramification indices of the places of F/k lying over the place P of the coefficient field $k(x)$ of F. The function field F must be a finite extension of $k(x)$.

`Zeros(a)`

A sequence containing all zeros of the algebraic function a.

`Poles(a)`

A sequence containing all poles of the algebraic function a.

`S ! I`
`Place(I)`

The place corresponding to the prime ideal I, where I is defined over the 'finite' or 'infinite' maximal order and S is the set of places of a function field.

`Support(D)`

Sequences containing the places and exponents occurring in the divisor D.

`AssignNames(~P, s)`

Change the print name employed when displaying P to be the first element in the sequence of strings s which must have length 1.

`InfinitePlaces(F)`

The infinite places of the function field F.

42.13.2.2 Global Function Field Places

F/k denotes a global function field in this section.

`HasPlace(F, m)`

Returns `true` and a place of degree m if and only if there exists such in the function field F/k; `false` otherwise.

`HasRandomPlace(F, m)`

Returns `true` and a random place of degree m in the function field F/k or (`false` if there are none).

`RandomPlace(F, m)`

Returns a random place of degree m in the function field F/k or throws an error if there is none.

`Places(F, m)`

A sequence containing the places of degree m of the function field F/k.

Example H42E33

Some creation of places is illustrated below.

```
> P<t> := PolynomialRing(Integers());
> N := NumberField(t^2 + 2);
> P<x> := PolynomialRing(N);
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(y^4 + x^5 - N.1^7);
> F;
Algebraic function field defined over Univariate rational function field over N
by
y^4 + x^5 + 8*N.1
> Zeros(c);
[ (x^5 + 8*N.1, c + x^5 + 8*N.1) ]
> P<y> := PolynomialRing(F);
> F2<d> := FunctionField(y^2 + F!N.1);
> Decomposition(F2, $1[1]);
[ (x^5 + 8*N.1, c + 2*x^5 + 16*N.1) ]
> DecompositionType(F2, $2[1]);
[ <2, 1> ]
> Places(F2)!$3[1];
(x^5 + 8*N.1, c + 2*x^5 + 16*N.1)
```

42.13.3 Related Structures

42.13.3.1 Parent and Category

The sets of function field places form the Magma category `PlcFun`. The notional power structure exists as parent but allows no operations.

`FunctionField(S)`

The corresponding function field of the set of places S.

`DivisorGroup(F)`

The group of divisors of the algebraic function field F/k, which is the free abelian group generated by the elements of the set of places of F/k.

42.13.4 Structure Invariants

42.13.4.1 General function fields

`WeierstrassPlaces(F)`

`SeparatingElement` FldFunGElt *Default :*

The Weierstrass places of the function field F/k. The semantics of calling `WeierstrassPlaces()` with F/k or the zero divisor of F/k are identical. See the description of `WeierstrassPlaces` on page 1168.

42.13.4.2 Global Function Fields

F/k denotes a global function field in this section.

`NumberOfPlacesOfDegreeOneOverExactConstantField(F, m)`

`NumberOfPlacesOfDegreeOneECF(F, m)`

The number of places of degree one in the constant field extension of degree m of the function field F/k. Contrary to the `Degree()` function the degree is here taken over the respective exact constant fields.

`NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(F, m)`

`NumberOfPlacesOfDegreeOneECFBound(F, m)`

The minimum of the Serre and Ihara bound on the number of places of degree one in the constant field extension of degree m of the function field F/k. Contrary to the `Degree()` function the degree is here taken over the respective exact constant fields.

`NumberOfPlacesOfDegreeOverExactConstantField(F, m)`

`NumberOfPlacesDegECF(F, m)`

The number of places of degree m of the function field F/k. Contrary to the `Degree()` function the degree is here taken over the respective exact constant fields.

42.13.5 Structure Predicates

`S1 eq S2` `S1 ne S2`

42.13.6 Element Operations

42.13.6.1 Parent and Category

`Parent(P)` `Category(P)`

42.13.6.2 Arithmetic Operators

`- P`

`P1 + P2` `P1 - P2` `k * P` `P div k` `P mod k`

`Quotrem(P, k)`

Returns divisors D_1, D_2 such that the place $P = kD_1 + D_2$ and the exponents in D_2 are of absolute value less than $|k|$. The operations `div` and `mod` yield D_1 resp. D_2.

42.13.6.3 Equality and Membership

`P1 eq P2` `P1 ne P2` `P in S` `P notin S`

42.13.6.4 Predicates on Elements

`IsFinite(P)`

Returns `true` if the place P is a 'finite' place.

`IsWeierstrassPlace(P)`

Whether the degree one place P is a Weierstraß place of its function field F. See the description of `WeierstrassPlaces` on page 1168.

42.13.6.5 Other Element Operations

`FunctionField(P)`

The function field that corresponds to the place P.

`Degree(P)`

The degree of the place P over the constant field of definition k.

`RamificationIndex(P)`
`RamificationDegree(P)`

The ramification index of the place P over its subplace of the rational function field $k(x)$ (the function field of P must be a finite extension of $k(x)$).

`InertiaDegree(P)`
`ResidueClassDegree(P)`

The degree of inertia (or residue class degree) of a place P over the corresponding subplace of the rational function field (the function field of P must be a finite extension of $k(x)$)

`Minimum(P)`

A monic prime polynomial in $k[x]$ or $1/x$ or an ideal, corresponding to the place of the coefficient field of the function field of the place P which P lies above (the function field of P must be a finite extension of $k(x)$).

`ResidueClassField(P)`

The residue class field of the place P and the map from the order of the place into the field.

`Evaluate(a, P)`

Evaluate the algebraic function a at the place P. If it is not defined at P, infinity is returned.

`Lift(a, P)`

Lift the element a of the residue class field of the place P (including infinity) to an algebraic function.

`TwoGenerators(P)`

Two algebraic functions having the place P as their unique common zero.

`LocalUniformizer(P)`

`UniformizingElement(P)`

A local uniformizing parameter at the place P.

`SafeUniformizer(P)`

The valuation of the element a at the place P.

`Ideal(P)`

Create a prime ideal corresponding to the place P.

`Norm(P)`

The divisor of the norm of the ideal of the place P.

Example H42E34

```
> R<x> := FunctionField(GF(9));
> P<y> := PolynomialRing(R);
> f := y^4 + (2*x^5 + x^4 + 2*x^3 + x^2)*y^2 + x^8
>      + 2*x^6 + x^5 +x^4 + x^3 + x^2;
> F<a> := FunctionField(f);
> Genus(F);
7
> NumberOfPlacesDegECF(F, 2);
28
> P := RandomPlace(F, 2);
> P;
(x^2 + $.1^2*x + $.1^7, a + $.1^5*x + $.1^5)
> LocalUniformizer(P);
x^2 + $.1^2*x + $.1^7
> TwoGenerators(P);
x^2 + $.1^2*x + $.1^7 a + $.1^5*x + $.1^5
> ResidueClassField(P);
Finite field of size 3^4
> Evaluate(1/LocalUniformizer(P), P);
Infinity
> Valuation(1/LocalUniformizer(P), P);
-1
```

42.13.7 Completion at Places

`Completion(F, p)`

`Completion(O, p)`

`Precision` RngIntElt *Default :* 20

The completion of the algebraic function field F or an order O of such at the place p of F or the function field of O. The map from F or O into the series ring is returned also.

The series ring returned is an infinite precision ring whose default precision for elements is given by the `Precision` parameter.

42.14 Divisors

42.14.1 Creation of Structures

`DivisorGroup(F)`

Create the group of divisors of the algebraic function field F/k.

42.14.2 Creation of Elements

`Divisor(P)`

`Div ! P`

`1 * P`

Given a place P in a function field, return the prime divisor $1 * P$.

`Div ! a`

`Divisor(a)`

Given an algebraic function a, return the principal divisor (a).

`Div ! I`

`Divisor(I)`

The divisor corresponding to the factorization of the ideal I.

`Divisor(I, J)`

The divisor corresponding to the ideal factorization of the ideals I and J belonging to the 'finite' and 'infinite' maximal order.

`Identity(G)`

`Id(G)`

Given the group G of divisors of a function field, return the zero divisor.

`CanonicalDivisor(F)`

A canonical divisor of the function field F/k.

`DifferentDivisor(F)`

The different divisor of the underlying extension of the function field $F/k(x)$.

`AssignNames(~D, s)`

Change the print name employed when displaying D to be the contents of s which must have length 1 in this case.

42.14.3 Related Structures

42.14.3.1 Parent and Category

The group of divisors form the Magma category `DivFun`. The notional power structure exists as parent but allows no operations.

`FunctionField(G)`

Given the group G of divisors of a function field F/k, return F.

`Places(F)`

The set of places of the algebraic function field F/k.

42.14.4 Structure Invariants

`NumberOfSmoothDivisors(n, m, P)`

The number of effective divisors of degree less equal n who consist of places of degree less equal m only. The sequence element $P[i]$ contains the (generic) number of places of degree $1 \leq i \leq \min\{n, m\}$. The formula used is described in [Heß99].

`DivisorOfDegreeOne(F)`

A divisor of degree one over the exact constant field of the global function field F/k.

42.14.5 Structure Predicates

`Div1 eq Div2` `Div1 ne Div2`

42.14.6 Element Operations

42.14.6.1 Arithmetic Operators

`- D`

`D1 + D2` `D1 - D2` `k * D` `D div k` `D mod k`

`P + D` `D + P` `D - P` `P - D`

`Quotrem(D, k)`

Returns divisors D_1, D_2 such that the divisor $D = kD_1 + D_2$ and the exponents in D_2 are of absolute value less than $|k|$. The operations `div` and `mod` yield D_1 resp. D_2.

`GCD(D1, D2)`

`Gcd(D1, D2)`

`GreatestCommonDivisor(D1, D2)`

The greatest common divisor of the divisors $D1$ and $D2$.

`LCM(D1, D2)`

`Lcm(D1, D2)`

`LeastCommonMultiple(D1, D2)`

The least common multiple of the divisors $D1$ and $D2$.

42.14.6.2 Equality, Comparison and Membership

`D1 eq D2` `D1 ne D2`

`D1 le D2` `D1 lt D2` `D1 ge D2` `D1 gt D2`

`D in Div` `D notin Div`

42.14.6.3 Predicates on Elements

`IsZero(D)`

`IsEffective(D)` `IsPositive(D)`

`IsSpecial(D)` `IsPrincipal(D)`

`IsCanonical(D)`

Returns `true` iff the divisor D is canonical and a differential having D as its divisor.

Example H42E35

We show some simple creations and operations on divisors.

```
> PF<x> := PolynomialRing(GF(13, 2));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b>;
> CanonicalDivisor(FF2);
Complementary divisor of Divisor in ideal representation:
Ideal of Maximal Equation Order of FF2 over Maximal Equation Order of FF1 over
Univariate Polynomial Ring in x over GF(13^2)
Generator:
1, Fractional ideal of Maximal Order of FF2 over Maximal Equation Order of FF1
over Valuation ring of Univariate rational function field over GF(13^2) with
generator 1/x
Generator:
x^2
> IsCanonical($1);
true
> D := Divisor(b) + Divisor(d);
> E := Divisor(Random(FF2, 2)*MaximalOrderFinite(FF2),
> Random(FF2, 2)*MaximalOrderInfinite(FF2));
> d := D + E;
> d;
Divisor in reduced representation:
Dtilde :
Divisor in ideal representation:
Fractional ideal of Maximal Equation Order of FF2 over Maximal Equation Order of
FF1 over Univariate Polynomial Ring in x over GF(13^2)
Basis:
Pseudo-matrix over Maximal Equation Order of FF1 over Univariate Polynomial Ring
in x over GF(13^2)
Ideal of Maximal Equation Order of FF1 over Univariate Polynomial Ring in x over
GF(13^2)
Generator:
1 * ( 1 0 0 )
Ideal of Maximal Equation Order of FF1 over Univariate Polynomial Ring in x over
GF(13^2)
Generator:
1 * ( 0 1 0 )
Fractional ideal of Maximal Equation Order of FF1 over Univariate Polynomial
Ring in x over GF(13^2)
Generators:
1
($.1^137*x^12 + $.1^80*x^11 + $.1^22*x^10 + $.1^79*x^9 + $.1^88*x^8 +
   $.1^138*x^7 + $.1^130*x^6 + $.1^127*x^5 + $.1^163*x^4 + $.1^78*x^3 + 6*x^2 +
   $.1^41*x + $.1^146)/(x^12 + $.1^166*x^11 + $.1^50*x^10 + $.1^136*x^9 +
   $.1^32*x^8 + $.1^46*x^7 + $.1^134*x^6 + $.1^64*x^5 + 8*x^4 + $.1^93*x^3 +
   $.1^153*x^2 + $.1^162*x)*b + ($.1^24*x + $.1^153)/(x^11 + $.1^166*x^10 +
   $.1^50*x^9 + $.1^136*x^8 + $.1^32*x^7 + $.1^46*x^6 + $.1^134*x^5 +
   $.1^64*x^4 + 8*x^3 + $.1^93*x^2 + $.1^153*x + $.1^162) * ( $.1^161*x^11 +
   $.1^74*x^10 + $.1^145*x^9 + $.1^72*x^8 + $.1^122*x^7 + $.1^123*x^6 + 3*x^5 +
   $.1^133*x^4 + 2*x^3 + $.1^105*x^2 + $.1^102*x $.1^48*x^11 + $.1^82*x^10 +
   4*x^9 + $.1^102*x^8 + $.1^145*x^7 + $.1^118*x^6 + $.1^129*x^5 + $.1^102*x^4
   + $.1^138*x^3 + $.1^146*x^2 + $.1^134*x 1 ) , Ideal of Maximal Order of FF2
over Maximal Equation Order of FF1 over Valuation ring of Univariate rational
function field over GF(13^2) with generator 1/x
Basis:
Pseudo-matrix over Maximal Equation Order of FF1 over Valuation ring of
Univariate rational function field over GF(13^2) with generator 1/x
Ideal of Maximal Equation Order of FF1 over Valuation ring of Univariate
rational function field over GF(13^2) with generator 1/x
Generator:
1/x^2 * ( 1 0 0 )
Ideal of Maximal Equation Order of FF1 over Valuation ring of Univariate
rational function field over GF(13^2) with generator 1/x
Generators:
1/x^3
($.1^21*x^3 + $.1^86*x^2 + $.1^151*x + $.1^48)/x^6*b + $.1^79/x^3 * ( 0 1 0 )
Ideal of Maximal Equation Order of FF1 over Valuation ring of Univariate
rational function field over GF(13^2) with generator 1/x
Generator:
1/x^3*b * ( 0 0 1 ) ,
r : 0,
A :
Divisor in ideal representation:
Ideal of Maximal Equation Order of FF2 over Maximal Equation Order of FF1 over
Univariate Polynomial Ring in x over GF(13^2)
Generator:
1, Fractional ideal of Maximal Order of FF2 over Maximal Equation Order of FF1
over Valuation ring of Univariate rational function field over GF(13^2) with
generator 1/x
Generators:
x
x,
a :
(x)^-1 * (b)
```

A nicer (but potentially more expensive) way to print, would be to ensure the divisor had a representation as a linear combination of places and exponents.

```
> p, e := Support(d);
> d;
4*(x, (($.1^24*x + 9)*b + ($.1^133*x + $.1^117))*d^2 + (($.1^83*x + $.1^36)*b +
    ($.1^97*x + $.1^2))*d + ($.1^101*x + $.1^165)*b + $.1^108*x) + (x + $.1^102,
    (($.1^141*x + $.1^113)*b + ($.1^157*x + $.1^48))*d^2 + (($.1^94*x + $.1^92)*b
    + ($.1^167*x + $.1^79))*d + ($.1^36*x + $.1^85)*b + $.1^18*x + 6) + (x^2 +
    $.1^47*x + 8, (($.1^19*x^3 + $.1^155*x^2 + $.1^75*x + $.1^106)*b + (8*x^3 +
    $.1^131*x^2 + $.1^125*x + $.1^46))*d^2 + (($.1^86*x^3 + $.1^11*x^2 +
    $.1^141)*b + ($.1^94*x^3 + $.1^127*x^2 + 6*x + $.1^57))*d + ($.1^68*x^3 +
    $.1^82*x^2 + $.1^52*x + $.1^69)*b + $.1^95*x^3 + $.1^55*x^2 + $.1^30*x +
    $.1) + (x^8 + $.1^138*x^7 + $.1^91*x^6 + $.1^59*x^5 + $.1^25*x^4 +
    $.1^74*x^3 + 6*x^2 + $.1^153*x + 5, (($.1^86*x^10 + 12*x^9 + $.1^5*x^8 +
    $.1^7*x^7 + $.1^123*x^6 + $.1^8*x^5 + $.1^77*x^4 + $.1^43*x^3 + $.1^110*x^2
    + $.1^124*x + $.1^51)*b + ($.1^78*x^9 + $.1^105*x^8 + $.1^153*x^7 + 6*x^6 +
    $.1^142*x^5 + $.1^152*x^4 + $.1^54*x^3 + $.1^9*x^2 + $.1^43*x + $.1^37))*d^2
    + (($.1^63*x^10 + $.1^125*x^9 + $.1^156*x^8 + $.1^44*x^7 + $.1^27*x^6 +
    $.1^127*x^5 + $.1^160*x^4 + $.1^46*x^3 + 9*x^2 + 8*x + $.1^37)*b +
    ($.1^99*x^10 + $.1^119*x^9 + $.1^103*x^8 + $.1^25*x^7 + $.1*x^6 +
    $.1^114*x^5 + $.1^133*x^4 + $.1^34*x^3 + $.1^4*x^2 + $.1^40*x + $.1^71))*d +
    ($.1^86*x^10 + $.1^7*x^9 + $.1^142*x^8 + 4*x^7 + $.1^161*x^6 + 2*x^5 +
    $.1^17*x^4 + $.1^50*x^3 + $.1^100*x^2 + $.1^144*x + $.1^12)*b + $.1^31*x^10
    + $.1^40*x^9 + 8*x^8 + 9*x^7 + $.1^39*x^6 + $.1^120*x^5 + $.1^114*x^4 +
    $.1^116*x^3 + $.1^43*x^2 + $.1^103*x + $.1^93) - 15*(1/x, (($.1^114*x^2 +
    $.1^96*x + 12)/x^3*b + ($.1^153*x^2 + 4*x + 12)/x^3)*d^2 + (($.1^17*x^2 +
    $.1^124*x + 12)/x^3*b + ($.1^159*x^2 + $.1^124*x + 12)/x^3)*d + ($.1^159*x^2
    + 6*x + 12)/x^3*b + ($.1^21*x + 12)/x^2)
> g := GCD(D, E);
> l := LCM(D, E);
> g + l eq d;
true
> g le D;
true
> l ge E;
true
```

42.14.6.4 Other Element Operations

`FunctionField(D)`

Given a divisor D, return the function field.

`Degree(D)`

The degree of the divisor D over k, the constant field of definition.

`Support(D)`

A sequence containing the places occurring in the divisor D.

`Numerator(D)`
`ZeroDivisor(D)`

The numerator of the divisor D.

`Denominator(D)`

`PoleDivisor(D)`

The denominator of the divisor D.

`Ideals(D)`

Create two ideals of the 'finite' and 'infinite' maximal order respectively corresponding to the divisor D.

`Norm(D)`

The divisor of the norms of the ideals of the divisor D.

`FiniteSplit(D)`

`FiniteDivisor(D)`

`InfiniteDivisor(D)`

Split the divisor D into its finite and infinite part, returning either 2 divisors which are the sum of the finite places in D and the sum of the infinite places in D or the appropriate one of these.

`Dimension(D)`

The dimension of the Riemann-Roch space $\mathcal{L}(D)$ of the divisor D over k, the constant field of definition.

`IndexOfSpeciality(D)`

The index of speciality of the divisor D, which equals the dimension of $\mathcal{L}(W - D)$ where W is a canonical divisor.

`ShortBasis(D :`*parameters*`)`

`Reduction`	BoolElt	*Default :* `true`
`Simplification`	MonStgElt	*Default :* "*Full*"

Compute a basis for the Riemann-Roch space of D in short form:

Let $F = k(x, y)$ be an algebraic function field defined by $f(x, y) = 0$ over k. Given a divisor D of F/k this function returns a basis of the k-vector space

$$\mathcal{L}(D) = \{a \in F^{\times} \mid (a) \geq -D\} \cup \{0\}$$

in the short form

$$B = [\, b_1 \ldots , b_n \,],\ [\, d_1, \ldots , d_n \,]$$

with $b_i \in F^{\times}$ and $d_i \in \mathbf{Z}$ for all $1 \leq i \leq n$, where n denotes the degree in y of the defining equation f of F, such that

$$\mathcal{L}(D) = \left\{ \sum_{i=1}^{n} \lambda_i b_i \mid \lambda_i \in k[x] \text{ with deg } \lambda_i \leq d_i \text{ for } 1 \leq i \leq n \right\}.$$

The optional argument `Reduction` controls whether to use divisor reduction internally or not; it defaults to `true`. For small divisors this is sometimes faster.

The optional argument `Simplification` controls whether the resulting basis is simplified or not; it defaults to `"Full"`. Simplification sometimes is not insignificantly expensive and can be avoided by setting the parameter to `"None"`.

The algorithm is described in [Heß99].

`Basis(D :`*parameters*`)`

`Reduction`	BoolElt	*Default :* `true`
`Simplification`	MonStgElt	*Default :* *"Full"*

A sequence containing a basis of the Riemann-Roch space $\mathcal{L}(D)$, for the divisor D.

The optional argument `Reduction` controls whether to use divisor reduction internally or not; it defaults to `true`. For small divisors this is sometimes faster.

The optional argument `Simplification` controls whether the resulting basis is simplified or not; it defaults to `"Full"`. Simplification sometimes is not insignificantly expensive and can be avoided by setting the parameter to `"None"`.

`RiemannRochSpace(D)`

Given a function field F/k and a divisor D belonging to F/k, return a vector space V and a k-linear mapping $h : V \longrightarrow F$ such that V is isomorphic to the Riemann-Roch space $\mathcal{L}(D) \subset F$ under h.

`Valuation(D, P)`

The exponent of the place P in the divisor D.

`Reduction(D)`

`Reduction(D, A)`

Let D be a divisor. Denote the result of both functions by $\tilde{D}$, r, A and a (for the second function the input A always equals the output A). The divisor A has (must have) positive degree and the following holds:

(i) $D = \tilde{D} + rA - (a)$,

(ii) $\tilde{D} \geq 0$ and $\deg(\tilde{D}) < g + \deg(A)$ (over the exact constant field),

(iii) $\tilde{D}$ has minimal degree among all such divisors satisfying (i), (ii).

`GapNumbers(D, P)`

The sequence of gap numbers of the divisor D at P where P must be a place of degree one:

Let F/k be an algebraic function field, D a divisor and P a place of degree one. An integer $m \geq 1$ is a gap number of D at P if $\dim\bigl(D + (m-1)P\bigr) = \dim(D + mP)$ holds. The gap numbers m of D satisfy $1 \leq m \leq 2g-1-\deg(D)$ and their cardinality equals the index of speciality $i(D)$. `GapNumbers(D, P)` returns such a particular sequence. The sequences of gap numbers of D at various P are independent of constant field extensions for perfect k and are the same for all but a finite number of places P of degree one (consider e.g. k algebraically closed). If P is omitted in the function call, this uniform sequence is returned by `GapNumbers(D)`. The places P where D has different sequences of gap numbers are called Weierstraß places of D and are returned by `WeierstrassPlaces(D)`. In the above mentioned functions it is equivalent to replace D by either F or the zero divisor.

`GapNumbers(D)`

`SeparatingElement` FldFunGElt *Default :*

The sequence of global gap numbers of the divisor D. A separating element used internally for the computation can be specified, it defaults to `SeparatingElement(F)`. See the description of `GapNumbers` on page 1165.

Example H42E36

Consider the function field F defined by the curve of genus 7 defined by

$$y^4 + (2*x^5 + x^4 + 2*x^3 + x^2)*y^2 + x^8 + 2*x^6 + x^5 + x^4 + x^3 + x^2$$

We construct the function field $F/\mathbf{F}_9$ and compute the Riemann-Roch space corresponding to a certain divisor.

```
> k<w> := GF(9);
> R<x> := FunctionField(k);
> P<y> := PolynomialRing(R);
> f := y^4 + (2*x^5 + x^4 + 2*x^3 + x^2)*y^2 + x^8
>      + 2*x^6 + x^5 +x^4 + x^3 + x^2;
> F<a> := FunctionField(f);
> Genus(F);
7
> P1 := RandomPlace(F, 1);
> P2 := RandomPlace(F, 1);
> D := P1 - P2;
> D;
(1/x, w^7/x^7*a^3 + w^5/x^5*a^2 + w^3/x^2*a + w) - (x, 2/(x^4 + x^2 + 2*x)*a^3 +
    w^3/x*a^2 + (w^5*x^3 + w^3*x + w^7)/(x^3 + x + 2)*a + w^5)
> IsPrincipal(336*D);
true
> infty := Poles(F!x)[1];
> V, h := RiemannRochSpace(11*infty);
> V;
KModule V of dimension 5 over GF(3^2)
> h;
Mapping from: ModFld: V to FldFun: F
> B := h(Basis(V));
> B;
[
    x/(x^3 + x + 2)*a^3 + (2*x^4 + 2*x^3 + x)/(x^3 + x + 2)*a,
    1/(x^3 + x + 2)*a^3 + (2*x^3 + 2*x^2 + 1)/(x^3 + x + 2)*a,
    a^2 + 2*x^3 + 2*x^2,
    1/x*a^2 + 2*x^2 + 2*x,
    1
]
> (B[2] + 2*B[3])@@h;
(    0     1     2     0     0)
```

Example H42E37

As a trivial but illustrative example we consider the algebraic function field generated by $\sin(x)$ and $\cos(x)$ over Q and construct a single function $a(x)$ such that $\sin(x)$ and $\cos(x)$ can be expressed in terms of $a(x)$:

```
> Qc<c> := PolynomialRing(RationalField());
> Qcs<s> := PolynomialRing(Qc);
> F<s> := FunctionField(s^2 + c^2 - 1);
> c := F!c;
> Genus(F);
0
> Zeros(s);
[ (c - 1, s), (c + 1, s) ]
> Zeros(c-1);
[ (c - 1, s) ]
> P := Zeros(c-1)[1];
> Degree(P);
1
> Dimension(1*P);
2
> Basis(1*P);
[ 1/(c - 1)*s, 1 ]
> a := Basis(1*P)[1];
> Degree(a);
1
> MinimalPolynomial(a);
$.1^2 + (c + 1)/(c - 1)
> (a^2 - 1)/(a^2 + 1);
c
> a * ((a^2 - 1)/(a^2 + 1) - 1);
```

s

Example H42E38

Over $Q(i)$ the familiar identities

$$\cos(x) = (\exp(ix) + \exp(-ix))/2$$

$$\sin(x) = (\exp(ix) - \exp(-ix))/(2i).$$

hold. In Magma one can proceed as follows:

```
> Qx<x> := PolynomialRing(RationalField());
> k<i> := NumberField(x^2 + 1);
> kc<c> := PolynomialRing(k);
> kcs<s> := PolynomialRing(kc);
> F<s> := FunctionField(s^2 + c^2 - 1);
> c := F!c;
> Genus(F);
0
> e := c + i*s;
> ebar := c - i*s;
> Degree(e);
1
> c eq (e + ebar) / 2;
true
> s eq (e - ebar) / (2*i);
true
```

`RamificationDivisor(D)`

`SeparatingElement` FldFunGElt *Default :*

The ramification divisor of the divisor D (using `SeparatingElement` for the computation which defaults to `SeparatingElement(F)` for F/k the function field of D):

Let F/k be an algebraic function field, x a separating variable and D a divisor. The ramification divisor of D is defined to be

$$i(D)\,(W - D) + \big(W_x(D)\big) + \nu\,(dx),$$

where W is a canonical divisor of F/k, $W_x(D)$ is the determinant of the Wronksian matrix of D with respect to x and ν is the sum of the Wronskian orders of D with respect to x. It is effective and consists of the Weierstraß places of D. The constant field k is required to be exact.

`WeierstrassPlaces(D)`

`SeparatingElement` FldFunGElt *Default :*

The Weierstrass places of the divisor D (using `SeparatingElement` for the computation which defaults to `SeparatingElement(F)` for F/k the function field of D):

Let F/k be an algebraic function field, D a divisor and P a place of degree one. An integer $m \geq 1$ is a gap number of D at P if $\dim\big(D+(m-1)P\big) = \dim(D+mP)$ holds. The gap numbers m of D at P satisfy $1 \leq m \leq 2g-1-\deg(D)$ and their cardinality equals the index of speciality $i(D)$. The sequences of gap numbers of D are independent of constant field extensions for perfect k and are the same for all but a finite number of places P of degree one (consider e.g. k algebraically closed). The places P of degree one at which D has different sequences of gap numbers are called Weierstraß places of D.

This function returns a list of all places of F/k (having not necessarily degree one) which are lying below Weierstraß places of D viewed in $F\bar{k}/\bar{k}$ (k perfect). The constant field k is required to be exact. Note that if the characteristic of F is positive this function is currently quite slow for large genus because of `Differentiation()`.

`IsWeierstrassPlace(D, P)`

Given a divisor D and a degree 1 place P of a function field, return whether P is a weierstrass place of D.

`WronskianOrders(D)`

`SeparatingElement` FldFunGElt *Default :*

Let D be a divisor of an algebraic function field F/k with separating element x and let $v_1, \dots v_l$ be a basis of $\mathcal{L}(D)$. For the differentiation D_x with respect to x consider the successively smallest $\nu_1 \leq \dots \leq \nu_l \in \mathbf{Z}^{\geq 0}$ such that the rows $D_x^{(\nu_i)}(v_1), \dots, D_x^{(\nu_i)}(v_l)$, $1 \leq i \leq l$ are F-linearly independent. The numbers $\nu_1, \dots, \nu_l$ are the Wronskian orders of D with respect to x and are returned. If D has dimension zero, the empty list is returned. The constant field k is required to be exact.

The separating element can be given by setting the `SeparatingElement` parameter appropriately.

`ComplementaryDivisor(D)`

Return the complementary divisor $D^{\#}$ of the divisor D. The function field F/k of D must be a finite extension of a rational function field $k(x)$. The divisor $D^{\#}$ equals $\mathrm{Diff}(F/k(x)) - D$ for F the function field of D and $\mathrm{Diff}(F/k(x))$ the different divisor of $F/k(x)$.

`DifferentialBasis(D)`

A basis of the space of differentials of the divisor D. See `DifferentialBasis` on page 1175 for details.

```
DifferentialSpace(D)
```

A vector space and the isomorphism from this space to the differential space of the divisor D.

```
Parametrization(F, D)
```

An element x in F which is a non constant element of the basis of the divisor D having degree one and a sequence of elements L in the rational function field are returned such that x generates the function field F over the constant field and L contains the images of the generators of F over its constant field in the rational function field.

42.14.7 Functions related to Divisor Class Groups of Global Function Fields

Let F/k be a global function field. The group of divisor classes is isomorphic to the product of a copy of $\mathbf{Z}$ and the group of divisors classes of degree zero which is a finite abelian group. Magma features an algorithm to compute the divisor class group by computing an abelian group G in the form $\mathbf{Z}/c_1\mathbf{Z} \times \ldots \times \mathbf{Z}/c_{2g}\mathbf{Z} \times \mathbf{Z}$ with integers $c_1 | \ldots | c_{2g}$ and a surjective homomorphism $f : Div(F) \to G$ from the divisor group to G whose kernel consists precisely of the principal divisors.

The algorithm employed is a randomized index calculus style method of expected subexponential running time for "small" constant field size and "large" genus. A description of this and other algorithms of this section can be found in [Heß99].

Elements in product representation may result from applying the maps returned by some of the computations below. It can be expensive to put these elements into sets and to test them for equality.

```
ClassGroupGenerationBound(q, g)
```

A bound B such that the places of degree (over the exact constant field) less than or equal to B, taken together with the places of a divisor of degree one, generate the whole divisor class group of any global function field of genus g over the exact constant field of q elements.

```
ClassGroupGenerationBound(F)
```

A bound B such that all places of degree (over the exact constant field) less than or equal to B, taken together with the places of a divisor of degree one, generate the whole divisor class group of the function field F. Particular properties of the function field are taken into account.

```
ClassNumberApproximation(F, e)
```

An approximation of the class number of the global function field F/k with multiplicative error less than $1 + e$ for $e > 0$. The formula

$$\left| \log\big(h \,/\, q^g\,\big) - \sum_{r=1}^{b} q^{-r}/r\big(N_r - (q^r + 1)\big) \right| \leq 2gq^{-b/2}/\big((q^{1/2} - 1)(b + 1)\big)$$

is used where N_r denotes the number of places of degree one in the constant field extension of degree r of F/k.

```
ClassNumberApproximationBound(q, g, e)
```

Returns an integer B such that all places of degree less than or equal to B of a global function field of genus g over the exact constant field of q elements have to be considered in order to approximate the class number with multiplicative error less than $1 + e$ for $e > 0$.

`ClassGroup(F :`*parameters*`)`

`DegreeBound`	RngIntElt	*Default :*
`SizeBound`	RngIntElt	*Default :*
`ReductionDivisor`	DivFunElt	*Default :*
`Proof`	BoolElt	*Default :*

The divisor class group of the function field F/k as an abelian group, a map of representatives from the class group to the divisor group and the homomorphism from the divisor group onto the divisor class group.

The optional parameter `DegreeBound` allows to control the size of the factor basis which consists of all places of degree less equal `DegreeBound` (plus a small additional amount; the degree is taken over the exact constant field). If not provided the algorithm tries to choose an appropriate value.

The optional parameter `SizeBound` bounds the size of the factor basis to not exceed `SizeBound` places. Every time the factor basis has to be enlarged during the computation it will be by no more than `SizeBound` additional places. If not provided there is no bound on the size of the factor basis. Every enlargement of the factor basis will append all places of the next degree.

The optional parameter `ReductionDivisor` contains the reduction divisor used in the relation search stage. Reasonable choices are divisors of small positive degree. If not provided the algorithm tries to choose an appropriate reduction divisor.

The optional parameter `Proof` indicates whether the computed result should be proven in a proof step. If a small degree bound for the factor basis is used and the divisor class group happens to be a product of a large number of cyclic groups the proof step can be very time consuming and `Proof := false` might be helpful. Once a value is given for `Proof` it remains the default value until set differently. The initial value of `Proof` for every function field is `true`.

`ClassGroupAbelianInvariants(F :`*parameters*`)`

`DegreeBound`	RngIntElt	*Default :*
`SizeBound`	RngIntElt	*Default :*
`ReductionDivisor`	DivFunElt	*Default :*
`Proof`	BoolElt	*Default :*

Computes a sequence of integers containing the Abelian invariants of the divisor class group of the function field F/k.

The optional parameters are the same as for `ClassGroup`.

`ClassNumber(F)`

The order of the group of divisor classes of degree zero of the function field F/k.

Example H42E39

Some class group calculations :

```
> Y<t> := PolynomialRing(Integers());
> R<x> := FunctionField(GF(9));
> P<y> := PolynomialRing(R);
> f := y^3 + y + x^5 + x + 1;
> F<alpha> := FunctionField(f);
> ClassNumberApproximation(F, 1.3);
24890.25505701632912193514
> ClassGroup(F);
Abelian Group isomorphic to Z/13 + Z/13 + Z/13 + Z/13 + Z
Defined on 5 generators
Relations:
    13*$.1 = 0
    13*$.2 = 0
    13*$.3 = 0
    13*$.4 = 0
Mapping from: Abelian Group isomorphic to Z/13 + Z/13 + Z/13 + Z/13 + Z
Defined on 5 generators
Relations:
    13*$.1 = 0
    13*$.2 = 0
    13*$.3 = 0
    13*$.4 = 0 to Divisor group of F
Mapping from: Divisor group of F to Abelian Group isomorphic to Z/13 + Z/13 +
Z/13 + Z/13 + Z
Defined on 5 generators
Relations:
    13*$.1 = 0
    13*$.2 = 0
    13*$.3 = 0
    13*$.4 = 0 given by a rule
> ClassNumber(F);
28561
> Evaluate(LPolynomial(F), 1);
28561
```

`GlobalUnitGroup(F)`

The group of global units of the function field F/k, i. e. the multiplicative group of the exact constant field, as an Abelian group, together with the map into F.

`IsGlobalUnit(a)`

Whether the function field element a is a global unit, i.e. a constant (equivalent to `IsConstant`).

`IsGlobalUnitWithPreimage(a)`

Returns `true` and the preimage of the function field element a in the global unit group, `false` otherwise.

`PrincipalDivisorMap(F)`

The map from the multiplicative group of the function field to the group of divisors.

`ClassGroupExactSequence(F)`

Returns the three maps in the center of the exact sequence

$$0 \to k^{\times} \to F^{\times} \to Div \to Cl \to 0$$

where $k^{\times}$ is the global unit group of the function field, $F^{\times}$ is the multiplicative group of the function field, Div is the divisor group and Cl is the divisor class group.

`SUnitGroup(S)`

The group of S-units as an Abelian group and the map into the function field, where S is a sequence of places of a function field.

`IsSUnit(a, S)`

Returns `true` if the function field element a is an S-unit for the sequence of places S, `false` otherwise.

`IsSUnitWithPreimage(a, S)`

Returns `true` and the preimage of the function field element a in the S-unit group if a is an S-unit for the sequence of places S, `false` otherwise.

`SRegulator(S)`

The S-Regulator for the sequence of places S.

`SPrincipalDivisorMap(S)`

The map from the multiplicative group of the function field to the group of divisors (mod places in the sequence S).

`IsSPrincipal(D, S)`

Returns `true` and a generator if the divisor D is principal modulo places in the sequence S, `false` otherwise

`SClassGroup(S)`

The S-class group for the sequence of places S as an Abelian group, a map of representatives from the S-class group to the group of divisors (mod places in S) and the homomorphism from the group of divisors (mod places in S) onto the S-class group.

`SClassGroupExactSequence(S)`

Returns the three maps in the center of the exact sequence

$$0 \to U(S) \to F^\times \to Div(S) \to Cl(S) \to 0$$

where $U(S)$ is the S-unit group, $F^\times$ is the multiplicative group of the function field, $Div(S)$ is the group of divisors (mod places in the sequence S) and $Cl(S)$ is the S-class group.

`SClassGroupAbelianInvariants(S)`

Computes a sequence of integers containing the Abelian invariants of the S-class group for the sequence of places S.

`SClassNumber(S)`

The order of the torsion part of the S-class group for the sequence of places S.

`ClassGroupPRank(F)`

Compute the p-rank of the class group of F/k where p is the characteristic of F/k. More precisely: Let F/k be a function field of characteristic p. Consider the subgroup $Cl^0(F/k)[p]$ of p-torsion elements of the group of divisor classes of degree zero. This function returns its dimension as an $\mathbf{F}_p$-vector space. Possible values range from 0 to g, where g is the genus of F/k. The field k is currently required to be a finite field.

`HasseWittInvariant(F)`

Return the Hasse–Witt invariant of F/k. More precisely: Let F/k be a function field of characteristic p. Let $F\bar{k}/\bar{k}$ be the constant field extension by the algebraic closure $\bar{k}$ of k within an algebraic closure $\bar{F}$ of F/k. Consider the subgroup $Cl^0(F\bar{k}/\bar{k})[p]$ of p-torsion elements of the group of divisor classes of degree zero. This function returns its dimension as an $\mathbf{F}_p$-vector space. Possible values range from 0 to g, where g is the genus of F/k. k is required to be perfect.

`TateLichtenbaumPairing(D1, D2, m)`

The Tate–Lichtenbaum pairing $Cl_0[m] \times Cl_0/mCl_0 \to k$ for coprime divisors $D1$ and $D2$.

Example H42E40

```
> k<w> := GF(9);
> R<x> := FunctionField(k);
> P<y> := PolynomialRing(R);
> f := y^4 + (2*x^5 + x^4 + 2*x^3 + x^2)*y^2 +
>            x^8 + 2*x^6 + x^5 + x^4 + x^3 + x^2;
> F<a> := FunctionField(f);
> D1 := Zeros(a)[1] - Poles(F!x)[1];
> D2 := Zeros(a)[4] - Poles(F!x)[2];
> G,mapfromG,maptoG:=ClassGroup(F : Proof:=false);
> Order(maptoG(D1));
48
> Order(maptoG(D2));
336
> TateLichtenbaumPairing(D1,D2,48);
w^7
> TateLichtenbaumPairing(D2,D1,336);
w^3
```

42.15 Differentials

Spaces of differentials of function fields can be created and the differentials belonging to them manipulated. Divisors can be created from differentials and modules generated by a collection of differentials can be formed.

42.15.1 Creation of Structures

`DifferentialSpace(F)`

The space of differentials of the algebraic function field F/k.

42.15.2 Creation of Elements

The simplest ways of creating a differential are given below.

`Differential(a)`

Create the differential $d(a)$ of the function field or order element a.

`Identity(D)`

The identity differential of the space of differentials D.

`IsCanonical(D)`

Returns `true` iff the divisor D is canonical and a differential having D as its divisor.

42.15.3 Related Structures

`FunctionField(D)`

The function field of the differentials in the space D.

`FunctionField(d)`

The function field of the differential d.

42.15.4 Subspaces

`SpaceOfDifferentialsFirstKind(F)`

`SpaceOfHolomorphicDifferentials(F)`

A vector space and the isomorphism from this space to the space of differentials of the first kind (holomorphic differentials) of the function field F/k.

`BasisOfDifferentialsFirstKind(F)`

`BasisOfHolomorphicDifferentials(F)`

A basis of the space of differentials of the first kind (holomorphic differentials) of the function field F/k.

`DifferentialBasis(D)`

Computes a basis for the space of differentials

$$\Omega(D) := \{ \omega \in \Omega(F/k) \mid (\omega) \geq D \}$$

for a divisor D of an algebraic function field F/k.

`DifferentialSpace(D)`

A vector space and the isomorphism from this space to the differential space of the divisor D.

Example H42E41

This example illustrates the differential space of a divisor and some of the operations that can be done with it.

```
> Q := Rationals();
> Qx<x> := PolynomialRing(Q);
> Qxy<y> := PolynomialRing(Qx);
> f1 := y^2 - (x-1)*(x-2)*(x-3)*(x-5)*(x-6);
> F := FunctionField(f1);
> d := Divisor(F.1) + Divisor(F!BaseRing(F).1);
> V1 := DifferentialSpace(d);
> d := 2*Divisor(F.1) - Divisor(F!BaseRing(F).1);
> V2 := DifferentialSpace(d);
> V1;
```

```
KModule V1 of dimension 2 over Rational Field
> V2;
KModule V2 of dimension 2 over Rational Field
> V1 meet V2;
KModule of dimension 0 over Rational Field
> D := DifferentialSpace(F);
> v := V1 ! [2/9, 4/9]; v;
V1: (2/9 4/9)
> D!v;
(2/9*x^2 + 4/9*x) d(x)
> V1!$1;
V1: (2/9 4/9)
> BasisOfDifferentialsFirstKind(F);
[ (x/(x^5 - 17*x^4 + 107*x^3 - 307*x^2 + 396*x - 180)*F.1) d(x), (1/(x^5 -
    17*x^4 + 107*x^3 - 307*x^2 + 396*x - 180)*F.1) d(x) ]
```

42.15.5 Structure Predicates

`D1 eq D2`

Return `true` if the spaces of differentials $D1$ and $D2$ are the same.

42.15.6 Operations on Elements

A number of general operations for elements are also provided for differentials as well as a number of specific functions for differentials.

42.15.6.1 Arithmetic Operators

`r * x`

`x * r`

`x + y`

`-x`

`x - y`

`x / y`

`x / r`

The operations on differentials are inherited from the vector space structure of the space of differentials. Additionally, this space is one–dimensional as a vector space over the function field itself. The quotient x/y of two differentials x and y then gives the unique $r \in F$ such that $x = ry$.

42.15.6.2 Equality and Membership

`x eq y`

Returns `true` if x and y are the same differential.

`x in D`

Returns `true` if x is in the space of differentials D.

42.15.6.3 Predicates on Elements

`IsExact(d)`

Return whether d is known to be an exact differential. If `true` additionally return a generator. If d is not already known to be exact then no attempts to determine whether d is exact or not are currently undertaken.

`IsZero(d)`

Return `true` if d is the zero differential.

42.15.6.4 Functions on Elements

`Valuation(d, P)`

The valuation of the differential d at the place P.

`Divisor(d)`

The divisor (d) of the differential d.

`Residue(d, P)`

The residue of the differential d at the place P, where P must (currently) have degree one.

Example H42E42

```
> PF<x> := PolynomialRing(GF(31, 3));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b : Check := false>;
> Differential(d);
(26/x*d) d(x)
> I := Random(FF2, 2)*MaximalOrderInfinite(FF2);
> P := Place(Factorization(I)[1][1]);
> Valuation(Differential(d), P);
-2
> IsExact(Differential(Random(FF2, 2)));
true
```

`Module(L, R)`

IsBasis	BoolElt	*Default :* false
PreImages	BoolElt	*Default :* false

The R-module generated by the differentials in the sequence L as an abstract module, together with the map into the space of differentials. The resulting modules can be used for intersection and inner sum computations.

If the optional parameter `IsBasis` is set `true` the function assumes that the given elements form a basis of the module to be computed.

If the optional parameter `PreImages` is set `true` then the preimages of the given elements under the map are returned as the third return value.

Both optional parameters are mainly used to save computation time.

`Relations(L, R)`

`Relations(L, R, m)`

The module of R-linear relations between the differentials of the sequence L. The argument m is used for the following: Let the elements of L be $a_1, \ldots, a_n$, V be the relation module $\subseteq R^n$ and define $M := \{ \sum_{i=1}^{m} v_i a_i \mid v = (v_i)_i \in V \}$. The function tries to compute a generating system of V such that the corresponding generating system of M consists of "small" elements.

Example H42E43

This example shows some of the conversions and operations possible with the results of `Module`.

```
> Q := Rationals();
> Qx<x> := PolynomialRing(Q);
> Qxy<y> := PolynomialRing(Qx);
> f1 := y^2 - (x-1)*(x-2)*(x-3)*(x-5)*(x-6);
> F := FunctionField(f1);
> D := DifferentialSpace(F);
> M7 := Module([Differential(3*F.1)], FieldOfFractions(Qx));
> M8 := Module([Differential(F.1), Differential(F!BaseRing(F).1)],
> FieldOfFractions(Qx));
> M12 := M7 meet M8;
> M16 := M7 + M8;
> assert M12 subset M7;
> assert M12 subset M8;
> assert M12 subset M16;
> r := M7![&+[Random([-100, 100])/Random([1, 100])*x^i : i in [1 .. 5]]/
> &+[Random([-100, 100])/Random([1,
> 100])*x^i : i in [1 .. 5]] : j in [1 .. Dimension(M7)]];
> assert M7!D!r eq r;
> r;
M7: ((1/100*x^4 + x^3 - x^2 + 1/100*x - 1/100)/(x^4 - 1/100*x^3 - 1/100*x^2 +
    1/100*x - 1/100))
> D!r;
```

```
((3/40*x^8 + 162/25*x^7 - 20937/200*x^6 + 114873/200*x^5 - 279531/200*x^4 +
    304167/200*x^3 - 24321/40*x^2 + 303/20*x - 297/50)/(x^9 - 1701/100*x^8 +
    2679/25*x^7 - 30789/100*x^6 + 19891/50*x^5 - 3593/20*x^4 - 63/10*x^3 +
    883/100*x^2 - 144/25*x + 9/5)*F.1) d(x)
```

`Cartier(b)`

`Cartier(b, r)`

The result of applying the Cartier operator r times to b. More precisely, let F/k be a function field over the perfect field k, $x \in F$ be a separating variable and $b = g\,dx \in \Omega(F/k)$ with $g \in F$ be a differential. The Cartier operator is defined by

$$C(b) = \left(-d^{p-1}g/dx^{p-1}\right)^{1/p} dx.$$

This function computes the r-th iterated application of C to b.

42.15.6.5 Other

`CartierRepresentation(F)`

`CartierRepresentation(F, r)`

Compute a row representation matrix of the Cartier operator on a basis of the space of holomorphic differentials of F/k (applied r times). More precisely, let F/k be a function field over the perfect field k, $\omega_1, \ldots, \omega_g \in \Omega(F/k)$ be a basis for the holomorphic differentials and $r \in \mathbf{Z}^{\geq 1}$. Let $M = (\lambda_{i,j})_{i,j} \in k^{g\times g}$ be the matrix such that

$$C^r(\omega_i) = \sum_{m=1}^{g} \lambda_{i,m}\omega_m$$

for all $1 \leq i \leq g$. This function returns M and $(\omega_1, \ldots, \omega_g)$.

Example H42E44

An example of a trivial cartier action.

```
> PF<x> := PolynomialRing(GF(31, 3));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b : Check := false>;
> Cartier(Differential(d), 4);
(0) d(x)
> CartierRepresentation(FF2, 3);
Matrix with 0 rows and 0 columns
[]
```

42.16 Weil Descent

Weil Descent is a technique that can be applied to cryptosystem attacks amongst other things. The general idea is as follows. If C is a curve defined over field L and K is a subfield of L, the group of points of the Jacobian of C, $Jac(C)$, over L is isomorphic to the group of points of a higher-dimensional abelian variety, the *Weil restriction* of $Jac(C)$, over K. If a curve D over K can be found along with an algebraic homomorphism defined over K from $Jac(D)$ to this Weil restriction, it is usually possible to transfer problems about $Jac(C)(L)$ (like discrete log problems) to $Jac(D)(K)$. D will have larger genus than C, but we have descended to a smaller field. The advantage is that techniques like Index calculus, where the time complexity depends more strongly on the field size than the genus, can now be applied.

An important special case is where C is an ordinary elliptic curve over a finite field of characteristic 2. In this case, Artin-Schreier theory gives a very concrete algorithm for computing a suitable D. This is the GHS attack (see [Gau00] or the chapter by F. Heß in [Bla05]). There is a corresponding function `WeilDescent` in the Elliptic Curve chapter that works with curves rather than function fields.

The functions below implement the GHS Weil Descent in a more general characteristic p setting.

```
WeilDescent(E,k)
```

The main function. E must be an "Artin-Schreier" function field over finite field K, an extension of k, of the following form. If p is the characteristic of K, then the base field of E is a rational function field $K(x)$ and $E.1$, the generator of E over $K(x)$ has minimal polynomial of the form $y^p - y = c/x + a + b * x$ for $a, b, c \in K$. These parameters must satisfy $b, c \neq 0$ and at least one of the following conditions
i) $Trace_{K/k}(b) \neq 0$
ii) $Trace_{K/k}(c) \neq 0$
iii) $Trace_{K/\mathbf{F}_p}(a) = 0$

Then, if E_1 is the Galois closure of the separable field extension E/k and $[K : k] = n$, there is an extension of the Frobenius generator of $G(K/k)$ to an automorphism of E_1 which fixes x and is also of order n. If F is the fixed field of this extension, then it has exact constant field k and so it is the function field of a (geometrically irreducible) curve over k. This is the WeilDescent curve. It's algebraic function field F is the first return value.

Note that when $p = 2$, E is the function field of an ordinary elliptic curve in characteristic 2 (multiply the defining equation by x^2 and take xy as the new y variable) and conversely, any such elliptic curve can be put into this form in multiple ways.

The second return value is a map from the divisors of E to the divisors of F which represents the homomorphism $Jac(Curve(E))(K) \to Jac(Curve(F))(k)$. This map, however does not attempt any divisor reduction. For the characteristic 2 `WeilDescent` function on elliptic curves mentioned in the introduction, if F is hyperelliptic, the divisor map returned is the actual homomorphism from the elliptic curve (with function field E) to the Jacobian of the hyperelliptic curve with function field F. This may be more convenient for the user.

There are functions below that may be called to return the genus and other attributes of F before actually going through with its construction.

One important special case, worth noting here, is that when $p = 2$ and b or c is in k, then the degree of F is also 2 and so the descent curve is a hyperelliptic curve.

`ArtinSchreierExtension(c,a,b)`

A convenience function to generate the function field which is an extension of $K(x)$ by the (irreducible) polynomial $y^p - y - c/x - a - bx$ where K is a finite field which is the parent of a, b, c and p is the characteristic of K.

`WeilDescentDegree(E,k)`

With E and k as in the main `WeilDescent` function, returns the degree of the descended function field F over its rational base field $k(x)$. The computation only involves the Frobenius action on a, b, c and the descent isn't actually performed.

`WeilDescentGenus(E,k)`

With E and k as in the main `WeilDescent` function, returns the genus of the descended function field F over its rational base field $k(x)$. The computation only involves the Frobenius action on a, b, c and the descent isn't actually performed.

`MultiplyFrobenius(b,f,F)`

This is a simple convenience function, useful for generating elements of K on which the Frobenius has a given minimal polynomial when considered as an $\mathbf{F}_p$-linear map (see the example below).

The argument b should be an element of a ring R, f a polynomial over ring R_0, which is naturally a subring of R and F a map from R to itself.

If $f = a_0 + a_1 x + \ldots$ then the function returns $f(F)(b)$ which is $a_0 + a_1 F(b) + a_2 F(F(b)) + \ldots$.

Example H42E45

We demonstrate with an example of descent over a degree 31 extension in characteristic 2, which results in a genus 31 hyperelliptic function field, and a small characteristic 3 example.

```
> SetSeed(1);
> k<u> := GF(2^5);
> K<w> := GF(2^155);
> Embed(k, K);
> k2<t> := PolynomialRing(GF(2));
> h := (t^31-1) div (t^5 + t^2 + 1);
> frob := map< K -> K | x :-> x^#k >;
> b := MultiplyFrobenius(K.1,h,frob);
> E := ArtinSchreierExtension(K!1, K!0, b);
> WeilDescentGenus(E, k);
```

```
31
> WeilDescentDegree(E,k);
2
> C,div_map := WeilDescent(E, k);
> C;
Algebraic function field defined over Univariate rational function field over
GF(2^5) by
y^2 + u*y + u^18/($.1^32 + u^29*$.1^16 + u^14*$.1^8 + u^2*$.1^4 + $.1^2 +
    u^9*$.1 + u^5)
> // get the image of the place representing a K-rational point
> pl := Zeroes(E!(BaseField(E).1)-w)[1];
> D := div_map(pl);
> Degree(D); //31*32
992
> k := GF(3);
> K<w> := GF(3,4);
> a := w+1;
> c := w;
> b := c^3+c;
> E := ArtinSchreierExtension(c, a, b);
> WeilDescentDegree(E,k);
27
> WeilDescentGenus(E,k);
78
> C := WeilDescent(E, k);
> C;
Algebraic function field defined over Univariate rational function field over
GF(3) by
y^27 + 2*y^9 + y^3 + 2*y + (2*$.1^18 + 2*$.1^12 + 2*$.1^9 + 2)/($.1^9 + $.1^3 +
    1)
```

42.17 Function Field Database

An optional database of function fields may be downloaded from the MAGMA website. This section defines the interface to that database. Each database is associated to a given finite field and extension degree. The supported combinations are:

$\mathbf{F}_2$: degrees 2, 3

$\mathbf{F}_3$: degree 2

$\mathbf{F}_4$: degrees 2, 3, 5

$\mathbf{F}_5$: degrees 2, 3, 4, 8

$\mathbf{F}_7$: degree 9

$\mathbf{F}_{11}$: degree 3

$\mathbf{F}_{13}$: degree 3

For each function field in the database, the following information is stored and may be used to limit the function fields of interest via the `sub` constructor: The genus; the number of degree one places; the class number; and the class group.

42.17.1 Creation

`FunctionFieldDatabase(q, d)`

Returns a database object for the function fields of degree d over $\mathbf{F}_q$.

`sub< D | : parameters >`

`Genus`	RngIntElt	*Default :*
`NumberOfDegreeOnePlaces`	RngIntElt	*Default :*

Returns a sub-database of D, restricting (or further restricting, if D is already a sub-database of the full database) the contents to those function fields satisfying the specified conditions. Note that it is not possible to "undo" restrictions with this constructor — the results are always at least as limited as D is.

The parameter `Genus` may be used to restrict the search to fields with the specified genus.

The parameter `NumberOfDegreeOnePlaces` may be used to restrict the search to only those fields with the specified number of places of degree one.

42.17.2 Access

`BaseField(D)`

`CoefficientField(D)`

Returns the finite field underlying each function field in the database.

`Degree(D)`

Returns the degree of each function field in the database.

`#D`

`NumberOfFields(D)`

Returns the number of function fields stored in the database.

`FunctionFields(D)`

Returns the sequence of function fields stored in the database.

Example H42E46

The genus of a degree four function field is at most 6. We can see the distribution in a database by counting the size of appropriate sub-databases:

```
> D := FunctionFieldDatabase(5, 4);
> #D;
196380
> [ #sub<D |: Genus := g> : g in [0..6] ];
[ 60, 480, 960, 12960, 35040, 63120, 83760 ]
```

42.18 Bibliography

[**Bj94**] Johannes A. Buchmann and Hendrik W. Lenstra jr. Approximating rings of integers in number fields. *J. Théor. Nombres Bordx.*, 6(2):221–260, 1994.

[**Bla05**] I. Blake, G. Serrousi and N. Smart, editor. *Advances in Elliptic Curve Cryptography*, volume 317 of *LMS LNS*. Cambridge University Press, Cambridge, 2005.

[**Bue04**] D. Buell, editor. *ANTS VI*, volume 3076 of *LNCS*. Springer-Verlag, 2004.

[**CHM98**] John H. Conway, Alexander Hulpke, and John McKay. On transitive permutation groups. *LMS Journal of Computation and Mathematics*, 1:1–8, 1998.

[**FK**] C. Fieker and J. Klüners. Computational Galois Theory I: Invariants and Computations over **Q**. in preparation.

[**Gau00**] P. Gaudry, F. Heß and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *J. Cryptology*, 15(1):19–46, 2000.

[**Gei03**] Katharina Geißler. *Berechnung von Galoisgruppen über Zahl- und Funktionenkörpern.* PhD Thesis, TU-Berlin, 2003. available at URL:http://www.math.tu-berlin.de/˜kant/publications/diss/geissler.pdf.

[**GK00**] Katharina Geißler and Jürgen Klüners. The determination of Galois Groups. *J. Symbolic Comp.*, 30(6):653–674, 2000.

[**Heß99**] Florian Heß. *Zur Divisorenklassengruppenberechnung in globalen Funktionenkörpern.* Dissertation, Technische Universität Berlin, 1999. URL:http://www.math.tu-berlin.de/∼kant/publications/diss/diss_FH.ps.gz.

[**Heß04**] Florian Heß. An algorithm for computing isomorphisms of algebraic function fields. In Buell [Bue04], pages 263–271.

[**Klü02**] Jürgen Klüners. Algorithms for Function Fields. *Exp. Math.*, (101):171–181, 2002.

[**SM85**] Leonhard H. Soicher and John McKay. Computing Galois Groups over the rationals. *J. Number Th.*, 20:273–281, 1985.

[**Sta73**] Richard P. Stauduhar. The determination of Galois Groups. *Math. Comp.*, 27:981–996, 1973.

[**Sti93**] Henning Stichtenoth. *Algebraic function fields and codes.* Springer-Verlag, Berlin, 1993.

[**Sut**] Nicole Sutherland. Computing Galois Groups of Polynomials (especially over Function Fields of Prime Characteristic). in preparation.

[**Sut12**] Nicole Sutherland. Efficient Computation of Maximal Orders of Radical (including Kummer) Extensions. *Journal of Symbolic Computation*, 47:552–567, 2012.

[**Sut13**] Nicole Sutherland. Efficient Computation of Maximal Orders of Artin–Schreier Extensions. Accepted by Journal of Symbolic Computation, 2013.

[**vHKN11**] M. van Hoeij, J. Klüners, and A. Novocin. Generating Subfields. In Anton Leykin, editor, *Proceedings ISSAC 2011*, 2011.

43 CLASS FIELD THEORY FOR GLOBAL FUNCTION FIELDS

Chapter 43
CLASS FIELD THEORY FOR GLOBAL FUNCTION FIELDS

Global function fields admit a class field theory in the same way as number fields do (Chapter 39). From a computational point of view the main difference is the use of divisors rather than ideals and the availability in general of analytical methods; see Section 43.6.

Class field theory deals with the abelian extensions of a given field. In the number field case, all abelian extensions can be parameterized using more general class groups, in the case of global function fields, the same will be achieved using the divisor class group and extensions of it.

43.1 Ray Class Groups

The ray divisor class group Cl_m modulo a divisor m of a global function field K is defined via the following exact sequence:

$$1 \to k^\times \to O_m^\times \to \mathrm{Cl}_m \to \mathrm{Cl} \to 1$$

where O_m^* is the group of units in the "residue ring" mod m, k is the exact constant field of K and Cl is the divisor class group of K. This follows the methods outlined in [HPP97].

```
RayResidueRing(D)
```

Let $D = \sum n_i P_i$ be an effective divisor for some places P_i (so $n_i > 0$). The ray residue ring R is the product of the unit groups of the local rings:

$$R = O_m^\times := \prod (O_{P_i}/P_i^{n^i})^\times$$

where O_{P_i} is the valuation ring of P_i.

The map returned as the second return value is from the ray residue ring R into the function field and admits a pointwise inverse for the computation of discrete logarithms.

```
RayClassGroup(D)
```

Let D be an effective (positive) divisor. The ray class group modulo D is a quotient of the group of divisors that are coprime to D modulo certain principal divisors. It may be computed using the exact sequence:

$$1 \to k^\times \to O_m^\times \to \mathrm{Cl}_m \to \mathrm{Cl} \to 1$$

Note that in contrast to the number field case, the ray class group of a function field is infinite.

The map returned as the second return value is the map from the ray class group into the group of divisors and admits a pointwise inverse for the computation of discrete logarithms.

Since this function uses the class group of the function field in an essential way, it may be necessary for large examples to precompute the class group. Using `ClassGroup` directly gives access to options that may be necessary for complicated fields.

`RayClassGroupDiscLog(y, D)`

`RayClassGroupDiscLog(y, D)`

Return the discrete log of the place or divisor y in the ray class group modulo the divisor D. This is a version of the pointwise inverse of the map returned by `RayClassGroup`. The main difference is that using the intrinsics the values are cached, ie. if the same place (or divisor) is decomposed twice, the second time will be instantaneous.

A disadvantage is that in situations where a great number of discrete logarithms is computed for pairwise different divisors, a great amount of memory is wasted.

Example H43E1

We will demonstrate the creation of some ray class and ray residue groups. First we have to create a function field:

```
> k<w> := GF(4);
> kt<t> := PolynomialRing(k);
> ktx<x> := PolynomialRing(kt);
> K := FunctionField(x^3-w*t*x^2+x+t);
```

Now, to create some divisors:

```
> lp := Places(K, 2);
> D1 := 4*lp[2]+2*lp[6];
> D2 := &+Support(D1);
```

And now the groups:

```
> G1, mG1 := RayResidueRing(D1); G1;
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/2 + Z/2 + Z/2 +
Z/2 + Z/4 + Z/60 + Z/60
Defined on 10 generators
Relations:
    2*G1.1 = 0
    2*G1.2 = 0
    2*G1.3 = 0
    2*G1.4 = 0
    2*G1.5 = 0
    2*G1.6 = 0
    2*G1.7 = 0
    4*G1.8 = 0
    60*G1.9 = 0
    60*G1.10 = 0
> G2, mG2 := RayResidueRing(D2); G2;
Abelian Group isomorphic to Z/15 + Z/15
Defined on 2 generators
Relations:
    15*G2.1 = 0
    15*G2.2 = 0
```

$G_1 = O_{D_1}^\times$ should surject onto $D_2 = O_{D_2}^\times$ since $D_2 | D_1$:

```
> h := hom<G1 -> G2 | [G1.i@mG1@@mG2 : i in [1..Ngens(G1)]]>;
> Image(h) eq G2;
true
> [ h(G1.i) : i in [1..Ngens(G1)]];
[
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    G2.2,
    G2.1
]
```

Ray class groups are similar and can be mapped in the same way:

```
> R1, mR1 := RayClassGroup(D1);
> R2, mR2 := RayClassGroup(D2); R2;
Abelian Group isomorphic to Z/5 + Z/15 + Z
Defined on 3 generators
Relations:
    5*R2.1 = 0
    15*R2.2 = 0
> hR := hom<R1 -> R2 | [R1.i@mR1@@mR2 : i in [1..Ngens(R1)]]>;
> Image(hR);
Abelian Group isomorphic to Z/5 + Z/15 + Z
Defined on 3 generators in supergroup R2:
    $.1 = R2.1
    $.2 = R2.2
    $.3 = R2.3
Relations:
    5*$.1 = 0
    15*$.2 = 0
> $1 eq R2;
```

```
true
```

Note that the missing C_3 part in comparing G_2 and R_2 corresponds to factoring by $k^\times$. The free factor comes from the class group.
Now, let us investigate the defining exact sequence for R_2:

```
> C, mC := ClassGroup(K);
> h1 := map<K -> G2 | x :-> (K!x)@@mG2>;
> h2 := hom<G2 -> R2 | [ G2.i@mG2@@mR2 : i in [1..Ngens(G2)]]>;
> h3 := hom<R2 -> C | [ R2.i@mR2@@mC : i in [1..Ngens(R2)]]>;
> sub<G2 | [h1(x) : x in k | x ne 0]> eq Kernel(h2);
> Image(h2) eq Kernel(h3);
> Image(h3) eq C;
```

So indeed, the exact sequence property holds.

43.2 Creation of Class Fields

Since the beginning of class field theory one of the core problems has been to find defining equations for the fields. Although most of the original proofs are essentially constructive, they rely on complicated and involved computations and thus were not suited for hand computations.

The method used here to compute defining equations is essentially the same as the one used for number fields. The main differences are due to the problem of p-extensions in characteristic p where Artin-Schreier-Witt theory is used, and the fact that the divisor class group is infinite.

`AbelianExtension(D, U)`

Given an effective divisor D and a subgroup U of the ray class group, (see `RayClassGroup`), Cl_D of D such that the quotient Cl_D/U is finite, create the extension defined by this data. Note that, at this point, no defining equations are computed.

`MaximalAbelianSubfield(K)`

For a relative extension K/k of global function fields, compute the maximal abelian subfield $K/A/k$ of K/k as an abelian extension of k. In particular, this function compute the norm group of K/k as a subgroup of a suitable ray class group.

`HilbertClassField(K, p)`

For a global function field K and a place p of K, compute the (a) Hilbert class field of K as an abelian extension of K. This field is characterised by being the maximal abelian unramified extension of K where p is totally split.

`FunctionField(A)`

`WithAut`	BoolElt	*Default :* `false`
Verbose	`ClassField`	*Maximum :* 3

Given an abelian extension of function fields as created by `AbelianExtension`, compute defining equations for the corresponding (ray) class field.

More precisely: Let $\mathrm{Cl}/U = \prod \mathrm{Cl}/U_i$ be a decomposition of the norm group of A such that the Cl/U_i are cyclic of prime power order. Then for each U_i the function will compute a defining equation, thus A is represented by the compositum.

If `WithAut` is `true`, the second sequence returned contains a generating automorphism for each of the fields returned as the first return value. If `WithAut` is `false`, the first (and only) return value is a function field in non-simple representation.

`MaximalOrderFinite(A)`

`MaximalOrderInfinite(A)`

Compute a finite or an infinite maximal order of the function field of the abelian extension A.

Example H43E2

First we have to create a function field, a divisor and a ray class group:

```
> k<w> := GF(4);
> kt<t> := PolynomialRing(k);
> ktx<x> := PolynomialRing(kt);
> K := FunctionField(x^3-w*t*x^2+x+t);
> lp := Places(K, 2);
> D := 4*lp[2]+2*lp[6];
> R, mR := RayClassGroup(D);
```

Let us compute the maximal extension of exponent 5 such that the infinite place is totally split. This means that we

- have to create a subgroup of R containing the image of the infinite place
- compute the fifth power of R
- combine the two groups

and use this as input to the class field computation:

```
> inf := InfinitePlaces(K);
> U1 := sub<R | [x@@ mR : x in inf]>;
> U2 := sub<R | [5*R.i : i in [1..Ngens(R)]]>;
> U3 := sub<R | U1, U2>;
> A := AbelianExtension(D, U3);
> A;
Abelian extension of type [ 5 ]
Defined modulo 4*(t^2 + t + w, $.1 + w^2*t + 1) +
2*(t^2 + w^2*t + w^2, $.1 + w*t + w)
over Algebraic function field defined over
```

```
Univariate rational function field over GF(2^2) by
x^3 + w*t*x^2 + x + t
> FunctionField(A);
Algebraic function field defined over K by
$.1^5 + (w*K.1^2 + (w*t + w^2)*K.1 + (w^2*t^2 + t
    + w^2))*$.1^3 + ((t^2 + 1)*K.1^2 + w*t*K.1 +
    (w*t^4 + w))*$.1 + (t^3 + w*t^2 + w^2*t)*K.1^2
    + (w*t^4 + w^2*t^2 + w^2)*K.1 + t^4 + w^2*t^3
    + w^2*t + w
> E := $1;
>  #InfinitePlaces(E);
10
```

Which shows that all the places in `inf` split completely.
Now, suppose we still want the infinite places to split, but now we are looking for degree 4 extensions such that the quotient of genus by number of rational places is maximal:

```
> q, mq := quo<R | U1>;
> l4 := [ x`subgroup : x in Subgroups(q : Quot := [4])];
> #l4;
768
> A := [AbelianExtension(D, x@@mq) : x in l4];
> s4 := [<Genus(a), NumberOfPlacesOfDegreeOne(a), a>
>                            : a in A];
> Maximum([x[2]/x[1] : x in s4]);
16/5 15
> E := FunctionField(s4[15][3]);
>  l22 := [ x`subgroup : x in Subgroups(q : Quot := [2,2])];
>  #l22;
43435
```

Since this is quite a lot, we won't investigate them here further.

43.3 Properties of Class Fields

Let D be an effective divisor and U be a subgroup of the ray class group Cl_D. The main existence theorem of class field theory asserts that there is exactly one function field corresponding to the quotient Cl_D/U whose Galois group is isomorphic to Cl_D/U in a canonical way.

Since the field is uniquely defined this way so are its invariants. Some of the invariants can easily be read off the groups involved. Therfore none of the functions listed here will compute a set of defining equations. They can therefore be used on very large fields.

`Conductor(m)`

Let m be an effective divisor. This function computes the conductor of Cl_m which is the smallest divisor f such that the projection $\mathrm{Cl}_m \to \mathrm{Cl}_f$ is surjective.

`Conductor(m, U)`

Let m be an effective divisor and U be a subgroup of the ray class group of m. This function computes the conductor of Cl_m/U which is the smallest divisor f such that the projection $\pi : \mathrm{Cl}_m/U \to \mathrm{Cl}_f/\pi(U)$ is an isomorphism.

`Conductor(A)`

For an abelian extension A of global function fields, compute its conductor, ie. the conductor of the norm group of A.

`DiscriminantDivisor(m, U)`

Let m be an effective divisor and U a subgroup of ray class group such that Cl_m/U is finite. The discriminant divisor is defined as the norm of the different divisor.

`DiscriminantDivisor(A)`

For an abelian extension A of a global function field, compute its discriminant divisor, ie. the norm of the different divisor. Note that the discriminant divisor can be computed from the norm group, thus no defining equation is derived.

`DegreeOfExactConstantField(m)`

Let m be an effective divisor. Since the ray class field modulo m is always an infinite field extension containing the algebraic closure of the constant field, this returns ∞.

`DegreeOfExactConstantField(m, U)`

Let m be an effective divisor and U be a subgroup of the ray class group, (see `RayClassGroup`), modulo m. This function computes the degree of the algebraic closure of the constant field in the class field corresponding to Cl_m/U. This can be infinite.

`DegreeOfExactConstantField(A)`

The degree of the exact constant field of the abelian extension A of a global function field. This is the degree of the algebraic closure of the constant field of the base field of A in A. The degree of this field can be computed from the norm group, thus no defining equation is derived.

`Genus(m, U)`

Let m be an effective divisor and U be a subgroup of the ray class group, (see `RayClassGroup`), modulo m. This function computes the genus of the class field corresponding to Cl_m/U.

`Genus(A)`

The genus of the abelian extension A of a global function field.

`DecompositionType(m, U, p)`

Let m be an effective divisor and U be a subgroup of the ray class group, (see `RayClassGroup`), modulo m such that the quotient Cl_m/U is finite. For a place p this function will determine the decomposition type of the place in the extension defined by Cl_m/U, i.e. it will return a sequence of pairs $\langle f, e\rangle$ containing the inertia degree and ramification index for all places above p.

`DecompositionType(A, p)`

For an abelian extension A of a global function field k and a place p of k, compute the degree and ramification index of all places P lying above p.

`NumberOfPlacesOfDegreeOne(m, U)`

Let m be an effective divisor and U be a subgroup of the ray class group, (see `RayClassGroup`), modulo m such that the quotient Cl_m/U is finite. This function will compute the number of places of degree 1 that the class field corresponding to Cl_m/U has.

`NumberOfPlacesOfDegreeOne(A)`

For an abelian extension A of global function fields, compute the number of places of A that are of degree one over the constant field of the base field of A.

`Degree(A)`

For an abelian extension A of global function fields, return the degree of A over its base ring.

`BaseField(A)`

For an abelian extension A of global function fields, return the base field, ie, the global field k over which A was created as an extension.

`A eq B`

For two abelian extensions of the same base field, decide if they describe the same field, ie if the norm groups pulled back into a common over group, agree.

`A subset B`

For two abelian extensions of the same base field, test if the first is contained in the second. This is done by comparing the norm groups in a common over group, thus defining equations are not computed.

`A meet B`

Compute the intersection of two abelian extensions of the same base field as an abelian extension.

`A * B`

Compute the compositum of two abelian extensions of the same base field as an abelian extension. Since both fields are normal, the compositum is well defined and can be computed from the norm groups alone.

43.4 The Ring of Witt Vectors of Finite Length

The ring of Witt vectors of length n (type `RngWitt`) over a global function field K parametrizes the cyclic extensions of K of degree p^n where p is the characteristic of K. Witt vectors (type `RngWittElt`) can be defined over any ring with positive characteristic p. The ring of Witt vectors of length n will always be of characteristic p^n. In the case of Witt vectors over finite fields, they can be seen as isomorphic to finite quotients of unramified p-adic rings.

The functionality offered here is mainly motivated by the class field theory which deals with vectors of short length only.

The Witt rings are based on code developed by David Kohel.

`WittRing(F, n)`

Creates the ring of Witt vectors of length n where F must be a field of positive characteristic.

`W ! a`

Witt vectors of the Witt ring W can be constructed from a where a is

- an element of the same ring
- an integer
- an element of the base ring
- a sequence of length `Length(W)` whose universe can be changed to the base ring of W.

`BaseRing(W)`

`BaseField(W)`

The field of coefficients of the Witt ring W.

`Length(W)`

The length (dimension) of the elements of the Witt ring W.

`Eltseq(a)`

The list of coefficients of the Witt vector a.

`Unity(W)`

`Zero(W)`

The one resp. zero in the Witt ring W.

`W . 1`

The first non-trivial basis element of the Witt ring.

`x in W` `a eq b` `a - b` `- a` `a + b` `a * b` `a ^ n`

`FrobeniusMap(W)`

The Frobenius map on the Witt ring W which is defined to be the map sending vectors to vectors where every coefficient is raised to the pth power.

`FrobeniusImage(e)`

Computes the image of the Witt vector e under the Frobenius map.

`VerschiebungMap(W)`

The verschiebungs map of the Witt ring W, i.e. shift all coefficients one position to the right and pad with a zero in front.

`VerschiebungImage(e)`

Computes the image of the Witt vector e under the verschiebung map.

`Random(W)`

For finite Witt rings, i.e. Witt rings defined over finite fields, return a random element.

`Random(W, n)`

For Witt rings where the base field admits a random function with size restriction n.

`TeichmuellerSystem(R)`

A Techmüller system for the local ring R, ie. a system of representatives for the residue class field of R that is closed under multiplication.

`LocalRing(W)`

Any ring W of Witt vectors of finite length over a finite field is isomorphic to some unramified local ring. This intrinsic will create the corresponding local ring and the embedding into it.

`ArtinSchreierMap(W)`

Returns the map $x \mapsto F(x) - x$ where F is the Frobenius map of the Witt ring W.

`ArtinSchreierImage(e)`

This function computes the image of the Witt vector e under the Artin-Schreier map.

```
FunctionField(e)
```

WithAut	BoolElt	*Default :* true
Check	BoolElt	*Default :* false
Abs	BoolElt	*Default :* false

A Witt vector $e = (e_1, \ldots, e_n)$ of length n over k where e_1 is not in the image of the Artin-Schreier map $e_1 \notin \{x^p - x : x \in k\}$ defines a cyclic extension K/k of degree p^n. This function will compute K.

If `WithAut` is `true` in addition to K it will compute a generating automorphism and return it as second return value.

If `Abs` is `true`, the function will compute a K as a single step extension of k, otherwise, K will be constructed as a series of n Artin-Schreier extensions.

If `Check` is `true`, it will be verified that the extension is of degree p^n. In particular the restrictions on e_1 are tested.

43.5 The Ring of Twisted Polynomials

The ring of twisted polynomials plays a core role in the analytic side of class field theory of global function fields. Twisted polynomials can be viewed as additive polynomials where the multiplication is the composition of two polynomials. Alternatively, they are the ring of polynomials in the Frobenius automorphism of the base field as indeterminate and multiplication defined in terms of application of the corresponding endomorphism. Thus Twisted polynomials have two natural polynomial representations:

- as (arbitrary) polynomials in F, the Frobenius automorphism
- as additive polynomials, ie. as polynomials where the only terms are T^{q^i}.

In Magma the first representation is used (as the degrees are lower), but the second can always be obtained by converting to a polynomial.

Since every endomorphism in positive characteristic can be represented by an additive polynomial (or an additive power series), this section can also be viewed as making the endomorphism ring accessible.

In Magma the ring of twisted polynomials is of type `RngUPolTwst` and individual elements are of type `RngUPolTwstElt`. Twisted polynomial rings can be created over any ring of characteristic $p > 0$. They are left euclidean and therefore left PIR.

43.5.1 Creation of Twisted Polynomial Rings

```
TwistedPolynomials(R)
```

q	RngIntElt	*Default :* false

Given a ring R of characteristic $p > 0$, create the ring of twisted polynomials over R. If `q` is given it has to be a power of the characteristic p, it defaults to $q := p$. The multiplication in this ring $R < F >$ is defined by $rF = Fr^q$ for all $r \in R$. Elements if $R < F >$ are represented as polynomials in F.

43.5.2 Operations with the Ring of Twisted Polynomials

`Unity(R)`

For a ring R of twisted polynomials return the polynomial representing 1.

`Zero(R)`

For a ring R of twisted polynomials return the polynomial representing 0.

`R eq S`

For two rings of twisted polynomials R and S test if they are equal, that is if the underlying polynomial ring is considered equal by Magma. Generically that means to test if the base rings of R and S coincide.

`BaseRing(R)`

The coefficient ring of the ring R of twisted polynomials.

`R . i`

For a ring of twisted polynomials R and an integer i which should be 1, return the transcendental element of R.

43.5.3 Creation of Twisted Polynomials

Apart from creation from polynomials directly, twisted polynomial can also be created by specifying some finite $\mathbf{F}_q$-vector space where the corresponding additive polynomial will have its roots.

`AdditivePolynomialFromRoots(x, P)`

`InfBound`	RngIntElt	*Default :* 5
`Map`	Map	*Default :* id
`Class`	DivFunElt	*Default :* 0
`Limit`	RngIntElt	*Default :* ∞
`Scale`	RngElt	*Default :* `false`

An additive polynomial is characterized by the fact that its set of zeros forms an $\mathbf{F}_q$ vector space. Conversely, given an $\mathbf{F}_p$-vector space M in R there is essentially one additive polynomial that has M as its set of roots. Given a place P and a ring element x, this function computes the additive polynomial with roots $L($`InfBound`$P)$, evaluated at x, ie. the module M is a Riemann-Roch space. By choosing x to be for example a transcendental element in a polynomial ring, the actual additive polynomial can be computed. If the parameter `Map` is given, the elements of the Riemann-Roch space are first mapped by this map before the polynomial is computed, thus allowing the creation of polynomials over the completion of a function field. If the parameter `Limit` is given, the polynomial is reduced modulo $x^{\mathbf{Limit}}$. If `Class` is set to a non-zero divisor, instead of $L(nP)$, the Riemann-Roch space $L(nP+$`Class`$)$ is used. If "tt Scale is set to a ring element that is either compatible with elements of the Riemann-Roch space or with elements in the codomain of the map, the module is scaled as well, thus allowing for normalization.

`Random(F, n)`

For F the ring of twisted polynomials over a finite ring (ie. a ring that supports the generation of random elements, this function will return a polynomial of degree $n-1$ with randomly chosen coefficients.

Example H43E3

```
> Fq<w> := GF(4);
> k<t> := RationalFunctionField(Fq);
> R := TwistedPolynomials(k:q := 4);
> R![1,1];
T_4 + 1
> R![w*t, 1];
T_4 + w*t
> $2 * $1;
T_4^2 + (w*t^4 + 1)*T_4 + w*t
> $2 * $3;
T_4^2 + (w*t + 1)*T_4 + w*t
> p := Places(k, 1)[2];
> a := AdditivePolynomialFromRoots(PolynomialRing(k).1, p
>          :InfBound := 2);
> a;
T_4^3 + ($.1^96 + $.1^84 + $.1^81 + $.1^72 + $.1^69 + $.1^66 + $.1^60
+ $.1^57 + $.1^54 + $.1^51 + $.1^48 + $.1^45 + $.1^42 + $.1^39 +
$.1^36 + $.1^30 + $.1^27 + $.1^24 + $.1^15 + $.1^12 + 1)/$.1^96*T_4^2
+ ($.1^96 + $.1^93 + $.1^90 + $.1^87 + $.1^72 + $.1^69 + $.1^66 +
$.1^63 + $.1^33 + $.1^30 + $.1^27 + $.1^24 + $.1^9 + $.1^6 + $.1^3 +
1)/$.1^108*T_4 + ($.1^90 + $.1^87 + $.1^84 + $.1^81 + $.1^78 + $.1^60
+ $.1^57 + $.1^54 + $.1^51 + $.1^48 + $.1^42 + $.1^39 + $.1^36 +
$.1^33 + $.1^30 + $.1^12 + $.1^9 + $.1^6 + $.1^3 + 1)/$.1^108
> R, mR := RiemannRochSpace(2*p);
> b := Polynomial(a);
> [ Evaluate(b, mR(x)) eq 0 : x in R];
[ true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true ]
> AdditivePolynomialFromRoots(PolynomialRing(k).1, p:InfBound := 2,
>   Map := func<x|Expand(x, p:RelPrec := 100)>);
T_4^3 + ($.1^-96 + $.1^-84 + $.1^-81 + $.1^-72 + $.1^-69 + $.1^-66 +
$.1^-60 + $.1^-57 + $.1^-54 + $.1^-51 + $.1^-48 + $.1^-45 + $.1^-42 +
$.1^-39 + $.1^-36 + $.1^-30 + $.1^-27 + $.1^-24 + $.1^-15 + $.1^-12 +
1 + O($.1^4))*T_4^2 + ($.1^-108 + $.1^-105 + $.1^-102 + $.1^-99 +
$.1^-84 + $.1^-81 + $.1^-78 + $.1^-75 + $.1^-45 + $.1^-42 + $.1^-39 +
$.1^-36 + $.1^-21 + $.1^-18 + $.1^-15 + $.1^-12 + O($.1^-8))*T_4 +
$.1^-108 + $.1^-105 + $.1^-102 + $.1^-99 + $.1^-96 + $.1^-78 + $.1^-75
+ $.1^-72 + $.1^-69 + $.1^-66 + $.1^-60 + $.1^-57 + $.1^-54 + $.1^-51
+ $.1^-48 + $.1^-30 + $.1^-27 + $.1^-24 + $.1^-21 + $.1^-18 +
O($.1^-8)
```

43.5.4 Operations with Twisted Polynomials

`A + B` `A - B` `- A` `A * B` `A ^ n`

`A eq B` `IsZero(A)`

`LeadingCoefficient(F)`

For a twisted polynomial F, return the leading coefficient as an element of the coefficient ring.

`ConstantCoefficient(F)`

For a twisted polynomial F, return the constant coefficient as an element of the coefficient ring.

`Degree(F)`

For a twisted polynomial F, return its degree. The degree of the underlying additive polynomial is q times the degree of the twisted polynomial.

`Quotrem(F, G)`

For twisted polynomials F and G in the same ring, perform a right division with remainder: This function computes Q and R such that $F = Q * G + R$ and the degree of R is less than the degree of G. In general, unless the coefficient ring is algebraically closed or perfect, there is no left quotient, so the ring of twisted polynomials is a left-PID, but no right-PID.

`GCD(F, G)`

For twisted polynomials F and G in the same ring, compute the creates common right divisor of F and G, ie a twisted polynomial H such that $F = f_1H$ and $G = f_2H$ for some twisted polynomials f_1 and f_2 and such that H is monic of maximal degree.

`BaseRing(F)`

For a twisted polynomial F, return the coefficient ring of F, ie. the ring where all the coefficients of F are from.

`Polynomial(G)`

For a twisted polynomial G, return the corresponding additive polynomial by replacing the transcendental element F by T^q and in general F^i by T^{q^i} for $i = 0, \ldots,$ degree of G.

```
SpecialEvaluate(F, x)
```

For a twisted polynomial F, return the result of the evaluation of the corresponding additive polynomial at the point x.

```
SpecialEvaluate(F, x)
```

For a univariate polynomial F, return the evaluation of F at x. This function in particular is optimized for sparse polynomials (for example additive polynomials) and imprecise coefficients. In the general case, a call to `Evaluate` will be faster.

```
Eltseq(F)
```

For a twisted polynomial F, return a sequence containing its coefficients.

43.6 Analytic Theory

Probably the most significant difference between the class field theories for number fields and function fields is the fact that function fields allow an analytic description of abelian extensions in general where number fields (currently) only admit the analytical view for extensions of the rationals (cyclotomic fields) and imaginary quadratic fields (CM-theory).

The analytic description is based on Drinfeld-modules of rank 1 (or in the case of the rational function field the Carlitz-module). Informally, a Drinfeld module is a representation of some (infinite) maximal order of a function field into the ring of additive polynomials over some appropriate ring containing the original field. Similar to CM-theory, abelian extensions are then generated by adjoining torsion points under this action.

```
CarlitzModule(R, x)
```

For a rational function field $k = \mathbf{F}_q(t)$ and a polynomial f in $k[t]$, compute the image of f under the Carlitz module as an element in the ring of twisted polynomials R. Specifically, the Carlitz module is the representation induced by sending t to $F + t$ where F is the transcendental element of R, the Frobenius of k. As $\mathbf{F}_q[t]$ is freely generated by t, this defines a homomorphism (a representation) of $\mathbf{F}_q[t]$ into the twisted polynomials over k.

Example H43E4

We demonstrate how the Carlitz-module can be used to define abelian extensions of the rational function field.

```
> Fq<w> := GF(4);
> k<t> := RationalFunctionField(Fq);
> R<T> := TwistedPolynomials(k:q := 4);
```

Suppose we want to create the Ray-class field modulo $p := t^2+t+w$, ie we want to find an abelian extension unramified outside p and the infinite place.

```
> p := t^2+t+w;
> P := CarlitzModule(R, p);
> F := Polynomial(P);
> Factorisation(F);
[
    <T, 1>,
    <T^15 + (t^4 + t + 1)*T^3 + t^2 + t + w, 1>
]
> K := FunctionField($1[2][1]);
> a := Support(DifferentDivisor(K));
> a[1];
(t^2 + t + w, K.1)
> [ IsFinite(x) : x in a];
[ true, false, false, false, false, false ]
> RamificationIndex(a[1]);
15
```

So, this shows that the field has the ramification bahaviour we wanted. However, the Carlitz-module will give us more information: We will demonstrate that the automorphism group of K is isomorphic to the unit group of $\mathbf{F}_q[t]/p$ under this module.

```
> q, mq := quo<Integers(k)|p>;
> au := func<X|Evaluate(Polynomial(CarlitzModule(R, X)), K.1)>;
> [ <IsUnit(x), Evaluate(DefiningPolynomial(K), au(x@@mq)) eq 0>
>                   : x in q];
[ <true, true>, <true, true>, <true, true>, <true, true>, <true,
true>, <true, true>, <true, true>, <true, true>, <true, true>, <true,
true>, <true, true>, <true, true>, <true, true>, <true, true>, <true,
true>, <false, false> ]
```

Now we try to create the same field using the algebraic class field machinery:

```
> D, U := NormGroup(K);
> Conductor(D, U);
($.1^2 + $.1 + w) + (1/$.1)
```

So this also shows that K is ramified exactly at p and the infinite places and, since the multiplicity is one, tamely ramified at both places.

```
> A := AbelianExtension(D, U);
> F := FunctionField(A);
> F;
Algebraic function field defined over Algebraic function field defined
over GF(2^2) by
$.2 + 1 by
$.1^3 + ($.1^2 + $.1 + w)^2
$.1^5 + w^2*$.1^3 + w*$.1 + (w^2*$.1^4 + w^2*$.1^2 + 1)/($.1^6 + $.1^5
    + w^2*$.1^4 + $.1^3 + $.1^2 + w^2*$.1 + 1)
> HasRoot(Polynomial(K, DefiningPolynomials(F)[1]));
true K.1^10 + (t^2 + t + w)*K.1^4 + (t^2 + t + w)*K.1
> HasRoot(Polynomial(K, DefiningPolynomials(F)[2]));
true w/(t^2 + t + w)*K.1^12 + (w*t^2 + w*t + 1)/(t^2 + t + w)*K.1^6 +
    w*K.1^3
```

```
AnalyticDrinfeldModule(F, p)
```

For F a global function field and p a place, compute an algebraic description of "the" Drinfeld modudule of rank 1 defined for the ring of functions integral outside p. More precisely, let R be the ring of functions integral outside p, ie. functions havinf their only poles at R. In Magma this is represented by changing the representation of the function field so that p is the only infinite place. Then R becomes simply the finite maximal order. Futhermore, let C be the completion of F at p and $\tilde{C}$ be the closure of the algebraic closure of C, thus D will play the role of the complex numbers in this theory, while C is closely related to the reals. By means of mapping R to its image in $D \subset C$, we obtain a lattice Λ of rank 1. Related to this lattice are certain exponential functions (analytic functions where the set of zeros is Λ). The transformation behaviour of such functions can be used to define a map from R to the endomorphisms of C, represented as additive (twisted) polynomials. In particular, under suitable normalisation, the lattice Λ as defined above, the transformation behaviour can be realised over the Hilbert-class field of F. Since a Drinfeld module is uniquely defined by specifying a single image of a non-constant element of R, this function returns a non-constant (as first return value) and the image as a twisted polynomial over the Hilbert-class field as a second. In case the place is of degree one, the Drinfeld module will also be sign-normalized.

```
Extend(D, x, p)
```

Given D, the image of a non-constant element under a Drinfeld module for the ring R of functions integral outside p, an element x in R, compute the image of x.

Example H43E5

Although the code is not restricted to genus 1 (or 2) and even though hyperelliptic function fields can be handled in a more direct fashion, we will demonstrate the computation of a Drinfeld module on an elliptic curve.
We begin by defining a curve.

```
> k<w> := GF(4);
> kt<t> := PolynomialRing(k);
> kty<y> := PolynomialRing(kt);
> F := FunctionField(y^3+w*t^5+w*t);
> Genus(F);
1
> ClassNumber(F);
3
```

Now, to define a Drinfeld module, we need to single out an "infinite" place:

```
> p := InfinitePlaces(F)[1];
> Degree(p);
1
```

Since the place p is of degree 1, our Drinfeld module will be sign-normalised.

```
> A, D := AnalyticDrinfeldModule(F, p);
> H<h> := CoefficientRing(D);
> A;
w^2/(x + 1)*F.1
> D;
T_4^2 + ((w*x^5 + 1)/(x^5 + x)*$.1^2*h^2 + (w^2*x^5 + w^2*x^4 + w^2*x
+ 1)/(x^4 + x^3 + x^2 + x)*F.1^2*h + (x^4 + x^2 + w)/(x^2 +
1)*$.1^2)*T_4 + w^2/(x + 1)*F.1
```

So, the Drinfeld module is uniquely determined by the image of a single non-constant (A) which is chosen to have maximal valuation at p and sign 1.
The values of the Drinfeld module are defined over H which is the Hilbert-class field of F, ie. the maximal abelian unramified extension where p is completely split.

```
> Sign(A, p);
1
> Valuation(A, p);
-2
```

To compute the value of "the" Drinfeld module of a different element we use `Extend`:

```
> b := F!t;
> Extend(D, b, p);
w*T_4^3 + ((w*x^20 + w*x^19 + w*x^18 + w*x^17 + w*x^16 + w*x^15 +
w*x^14 + w*x^13 + w*x^12 + w*x^11 + w*x^10 + w*x^9 + w*x^8 + w*x^7 +
w*x^6 + w*x^5 + w*x^4 + w*x^3 + w*x^2 + w*x + w)/(x^4 + x^3 + x^2 +
x)*$.1^2*h^2 + (w^2*x^20 + w^2*x^19 + w^2*x^16 + w^2*x^15 + w^2*x^4 +
w^2*x^3 + w^2)/(x^3 + x^2 + x + 1)*$.1^2*h + (x^20 + x^18 + x^12 +
x^10 + w*x^4 + w*x^2 + w^2)/(x^2 + 1)*$.1^2)*T_4^2 + ((w*x^5 + w^2*x^3
+ w^2*x^2 + x + w)/(x^2 + 1)*$.1*h^2 + (w^2*x^5 + w^2*x^4 + x^3 +
w)/(x + 1)*$.1*h + (x^5 + w^2*x^3 + w*x^2 + w^2*x)*$.1)*T_4 + x
```

To compute a "Ray-class field" from here we can use the following:

```
> P := Places(F, 2)[1];
> a,b := TwoGenerators(P);
> GCD(Extend(D, a, p), Extend(D, b, p));
T_4^2 + ((w*x^5 + 1)/(x^5 + x)*$.1^2*h^2 + (w^2*x^5 + w^2*x^4 + w^2*x
+ 1)/(x^4 + x^3 + x^2 + x)*$.1^2*h + (x^4 + x^2 + w)/(x^2 +
1)*$.1^2)*T_4 + w^2/(x + 1)*$.1 + w^2
> Polynomial($1);
T_4^16 + ((w*x^5 + 1)/(x^5 + x)*F.1^2*h^2 + (w^2*x^5 + w^2*x^4 + w^2*x
    + 1)/(x^4 + x^3 + x^2 + x)*F.1^2*h + (x^4 + x^2 + w)/(x^2 +
    1)*F.1^2)*T_4^4 + (w^2/(x + 1)*F.1 + w^2)*T_4
> R := FunctionField($1 div Parent($1).1);
> R;
Algebraic function field defined over H by
T_4^15 + ((w*x^5 + 1)/(x^5 + x)*F.1^2*h^2 + (w^2*x^5 + w^2*x^4 + w^2*x
    + 1)/(x^4 + x^3 + x^2 + x)*F.1^2*h + (x^4 + x^2 + w)/(x^2 +
    1)*F.1^2)*T_4^3 + w^2/(x + 1)*F.1 + w^2
```

This should be an abelian extension over F, ramified only at p and P. So let's try to verify that:

```
> f := MinimalPolynomial(R.1, F);
> Degree(f);
45
> Ra := FunctionField(f:Check := false);
> NormGroup(Ra:Cond := 1*p+P);
(1/x, 1/x^2*F.1) + (x^2 + x + w^2, F.1 + x + 1)
Abelian Group isomorphic to Z
Defined on 1 generator in supergroup:
    $.1 = 2*$.1 + 3*$.2 + $.3 (free)
> D, sub_group := $1;
> W := AbelianExtension(D, sub_group);
> W;
Abelian extension of type [ 3, 15 ]
Defined modulo (1/x, 1/x^2*$.1) + (x^2 + x + w^2, $.1 + x + 1)
over Algebraic function field defined over Univariate rational
function field over GF(2^2) by
y^3 + w*t^5 + w*t
> FunctionField(W);
Algebraic function field defined over F by
$.1^3 + (x + 1)^-2 * (x^2 + x + w^2)^-1 * (F.1 + w*x + w) * (F.1 +
    w^2*x + w^2)
$.1^3 + (x) * (x + 1)^-4 * (F.1 + x + 1)^2 * (x^2 + x + w^2)^-1 * (F.1
    + w*x + w) * (F.1 + w^2*x + w^2)
$.1^5 + (1/(x^2 + 1)*F.1^2 + w^2/(x + 1)*F.1 + (x^2 + x + 1))*$.1^3 +
    (w/(x^2 + 1)*F.1^2 + w*x*F.1 + (x^4 + x^2 + 1))*$.1 + (x^2 + x +
    1)/(x^2 + 1)*F.1^2 + w^2*x*F.1 + x^4 + w^2*x^2 + w*x + w^2
> WW := $1;
>  [HasRoot(Polynomial(Ra, x)) : x in DefiningPolynomials(WW)];
[ true, true, true]
```

Exp(x,p)

InfBound	RngIntElt	*Default :* 5
Map	Map	*Default :* id
Class	DivFunElt	*Default :* 0
Limit	RngIntElt	*Default :* ∞
Scale	RngElt	*Default :* false

In Drinfeld's theory of elliptic modules, one associates an exponential function to a lattice. The transformation of this function under scaling of the lattice gives then rise to the "Drinfeld-module". This function computes an approximation to the exponential of the "standard-lattice". More precisely: let R be the ring of functions integral outside the place p and let C be the completion of the function field at p.

Considered as a subset of C, R is a 1-dimensional lattice Λ in Drinfelds sense. The exponential associated to this lattice is the function

$$\exp : z \to z \prod{}' (1 - z/l)$$

where the product is taken over all non-zero lattice points l. This function can be seen to be an additive analytic function.

For $n > 0$, let now $L(np)$ be the Riemann-Roch spaces and

$$f_n : z \to z \prod{}' (1 - z/l)$$

(the product is over the non-zero elements of $L(np)$). It can be seen that $f_n \to \exp$ as $n \to \infty$. This intrinsic computes f_n for n equal the value of `InfBound`. If `Limit` is given then the twised polynomial representing f_n is truncated at that term. If `Class` is given and contains the divisor d, then instead of $L(np)$ we use $L(np + d)$ which will approximate a (in general) non-isomorphic exponential coming from the lattice from the ideal representing the finite part of d.

If `Map` is given, then the map is applied to each element in $L(np)$ first, thus allowing to compute analytic approximations instead of alegebraic ones. Additionally, if `Scale` is given, the elements of $L(np)$ are multiplied by this value before the functions are formed, corresponding to a scaling of the lattice.

The exponential is evaluated at x, the first argument. Typically, x will be the transcendental element of a polynomial ring, a twisted polynomial ring or a power series ring.

`AnalyticModule(x, p)`

`InfBound`	RngIntElt	*Default :* 5
`Map`	Map	*Default :* id
`Class`	DivFunElt	*Default :* 0
`Limit`	RngIntElt	*Default :* ∞
`Scale`	RngElt	*Default :* `false`

Let Λ be the lattice as described for `Exp` above. By Drinfeld's theory, the exponential functions of Λ and $x\Lambda$ are related through some polynomial. This function computed the polynomial for x, which is "the" image of x under the Drinfeld module defined by Λ. The use of the parameters is as for `Exp` above.

`CanNormalize(F)`

Let F be a twisted polynomial, typically over a completion. This function tries to conjugate F so that the coefficients are integral with small valuations. On success, `true`, the new polynomial and the element used to normalise F is returned.

`CanSignNormalize(F)`

Let F be a twisted polynomial, typically over a completion. This function tries to conjugate F so that the highest coefficient is an element in the residue class field. On success, `true`, the new polynomial and the element used to normalise F is returned.

`AlgebraicToAnalytic(F, p)`

Given a non-trivial image F under a Drinfeld module with the "infinite place" p, compute a basis for a submodule of the lattice underlying F. The parameter `RelPrec` is used to limit the number of coefficients of the exponential that are reconstructed, thus it also limits the dimenstion of the submodule.

43.7 Related Functions

This section list some related functions that are either useful in the context of class fields for function fields or are necessary for their computation. They will most certainly change their appearance.

`StrongApproximation(m, S)`

`Strict`	BoolElt	*Default :* `false`
`Exception`	DivFunElt	*Default :* `false`
`Raw`	BoolElt	*Default :* `false`

Given an effective divisor m and a sequence S of pairs (Q_i, e_i) of places and elements, find an element a and a place Q_0 such that

$$v_{Q_i}(a - e_i) \geq v_{Q_i}(m),$$

and a is integral everywhere outside Q_i $(0 \leq i \leq n)$.

If `Exception` is not `false`, it has to be a place that will be used for Q_0.

If `Strict` is `true`, the element a will be chosen such

$$v_{Q_i}(a - e_i) = v_{Q_i}(m)$$

If `Raw` is `true`, different rather technical return values are computed that are used internally.

Example H43E6

We first have to define a function field and some places:

```
> k<w> := GF(4);
> kt<t> := PolynomialRing(k);
> ktx<x> := PolynomialRing(kt);
> K := FunctionField(x^3-w*t*x^2+x+t);
> lp := Places(K, 2);
```

We will now try to find an element x in K such that $v_{p_i}(x - e_i) \geq m_i$ for $p_i =$`lp[i]`, $m_i = i$ and random elements e_i:

```
> e := [Random(K, 3) : i in lp];
> m := [i : i in [1..#lp]];
> D := &+ [ m[i]*lp[i] : i in [1..#lp]];
> x := StrongApproximation(D, [<lp[i], e[i]> : i in [1..#lp]]);
> [Valuation(x-e[i], lp[i]) : i in [1..#lp]];
[ 1, 2, 3, 4, 5, 6 ]
```

Note, that we only required $\geq$ for the valuations, to enforce $=$ we would need to pass the `Strict` option. This will double the running time.

`NonSpecialDivisor(m)`

`Exception`	DivFunElt	*Default :*

Given an effective divisor m, find a place P coprime to m and an integer $r \geq 0$ such that $rP - m$ is a non special divisor and return r and P.

If `Exception` is specified, it must be an effective divisor n coprime to m. In this case the function finds $r > 0$ such that $rn - m$ is non special and returns r and n.

`NormGroup(F)`

`Cond`	DivFunElt	*Default :*
`AS`	RngWittElt	*Default :*
`Extra`	RngIntElt	*Default :* 5

Given a global function field, try to compute its norm group. The norm group is defined to be the group generated by norms of unramified divisors. This group can be related to a subgroup of some ray class group.

Provided F is abelian, this function will compute a divisor m and a sub group U of the ray class group modulo m such that F is isomorphic to the ray class field thus defined.

This function uses a heuristic algorithm. It will terminate after the size of the quotient by the norm group is less or equal than the degree for `Extra` many places.

If `Cond` is given, it must be an effective divisor that will be used as the potential conductor of F. Note: if `Cond` is too small, ie. a proper divisor of the true conductor, the result of this function will be wrong. However, if the conductor is not passed in, the discriminant divisor is used as a starting point. As this is in general far too large, the function will be much quicker if a better (smaller) starting point is passed in.

If `AS` is given, it must be a Witt vector e of appropriate length and F should be the corresponding function field. This allows a much better initial guess for the conductor than using the discriminant.

`Sign(a, p)`

Given a function a in some global function field and a place p such that a is integral at p (has non-negative valuation) return the sign of a, ie. the first non-zero coefficient if the expansion of a at p. The sign function is not unique. Magma choses a sign function when creating the residue class field map.

`ChangeModel(F, p)`

Given a global function field F and a place p, return a new function field G that is $\mathbf{F}_q$-isomorphic to F and has p as the only infinite place.

43.8 Enumeration of Places

In several situations one needs to loop over the places of a function field until either the one finds a place with special properties or until they generate a certain group. The functions listed here support this.

`PlaceEnumInit(K)`

`Coprime`	Any	*Default :*
`All`	BoolElt	*Default :* `false`

Initialises an enumeration process for places of the function field K. The enumeration process will loop over all irreducible polynomials of the underlying finite field and for each polynomial over all primes lying above it.

If `Coprime` is given, it should be either a set of places that should be ignored in the process or a divisor. In case a divisor is passed in only places coprime to the divisor will be returned.

If `All` is `false`, the infinite places won't be considered.

`PlaceEnumInit(P)`

`Coprime`	Any	*Default :*

Constructs an enumeration process for places starting at the place P.

If `Coprime` is given, it should be either a set of places that should be ignored in the process or a divisor. In case a divisor is passed in only places coprime to the divisor will be returned.

`PlaceEnumInit(K, Pos)`

`Coprime`	Any	*Default :*

Constructs an enumeration environment that starts at the place of the function field K indexed by Pos as returned from `PlaceEnumPosition`.

`PlaceEnumCopy(R)`

Copies the environment and the current state of the enumeration process for places R.

`PlaceEnumPosition(R)`

Returns a list of integers that acts as an index to the places as enumerated by the environment R.

`PlaceEnumNext(R)`

Returns the "next" place of the process R.

`PlaceEnumCurrent(R)`

Returns the current place pointed to by the environment R, i.e. the last place returned by `PlaceEnumNext`.

43.9 Bibliography

[**HPP97**] Florian Heß, Sebastian Pauli, and Michael E. Pohst. On the computation of the multiplicative group of residue class rings. *Math. Comp.*, 1997.

44 ARTIN REPRESENTATIONS

Chapter 44
ARTIN REPRESENTATIONS

44.1 Overview

An *Artin representation* is a complex representation of $\mathrm{Gal}(\bar{\mathbf{Q}}/\mathbf{Q})$ that factors through some finite quotient $\mathrm{Gal}(F/\mathbf{Q})$. In Magma, Artin representations are represented as characters of $\mathrm{Gal}(F/\mathbf{Q})$, not the actual modules. They are allowed to be virtual, except in the L-function machinery (see Chapter 127).

44.2 Constructing Artin Representations

`ArtinRepresentations(K)`

`f`	RngUPolElt	*Default :*
`Ramification`	BoolElt	*Default :* `false`
`FactorDiscriminant`	BoolElt	*Default :* `false`
`p0`	RngIntElt	*Default :*

Compute all irreducible Artin representations that factor through the normal closure F of the number field K.

The Galois group $G = \mathrm{Gal}(F/K)$ whose representations are constructed is represented as a permutation group on the roots of `f`, which must be a monic irreducible polynomial with integer coefficients that defines K. By default this is the defining polynomial of K represented as an extension of $\mathbf{Q}$. (It is possible to specify any monic integral polynomial whose splitting field is F, even a reducible one, but `PermutationCharacter(K)` and the Dedekind ζ-function of K will not work correctly.)

The `Ramification` parameter specifies whether to pre-compute the inertia groups at all ramified primes and the conductors of all representations.

The parameter `FactorDiscriminant` determines whether to factorize the discriminant of `f` completely, even if it appears to contain large prime factors. The factorization is used to determine which primes ramify in F/K, which is necessary to compute the conductors. If the factorization is incomplete, Magma assumes that the primes in the unfactored part of the discriminant are unramified. One may specify `FactorDiscriminant:=`

`<TrialLimit,PollardRhoLimit,ECMLimit,MPQSLimit,Proof>` and these 5 parameters are passed to the `Factorization` function; the default behaviour (`false`) is the same as `<10000,65535,10,0,false>`. When the factorization is incomplete, Magma will print "(?)" following the conductor values, when asked to print an Artin representation.

Finally, `p0` specifies which p-adic field to use for the roots of `f`, in particular in Galois group computations. It must be chosen so that `GaloisGroup(f:Prime:=p0)` is successful. By default it is chosen by the Galois group computation.

`K !! ch`

Writing F for the normal closure of $K/\mathbf{Q}$, this function converts an abstract group character of $\mathrm{Gal}(F/\mathbf{Q})$ or the sequence of its trace values into an Artin representation.

`PermutationCharacter(K)`

Construct the permutation representation A of the absolute Galois group of $\mathbf{Q}$ on the embeddings of K into $\mathbf{C}$. This is an Artin representation of $\mathrm{Gal}(F/\mathbf{Q})$ of dimension $[K:\mathbf{Q}]$, where F is the normal closure of K, and it is the same as the permutation representation of $\mathrm{Gal}(F/\mathbf{Q})$ on the cosets of $\mathrm{Gal}(F/K)$.

`Determinant(A)`

Construct the determinant of a given Artin representation. The result is given as a 1-dimensional Artin representation attached to the same field.

`ChangeField(A,K)`

`K !! A`

`MinPrimes` RngIntElt *Default :* 20

Given an Artin representation (attached to some number field) that is known to factor through the Galois closure of K, attempts to recognize it as such. Returns "the resulting Artin representation attached to K", `true` if successful, and 0, `false` if it proves that there is no such representation. The parameter `MinPrimes` specifies the number of additional primes for which to compare traces of Frobenius elements.

Example H44E1

A quadratic field K has two irreducible Artin representations the factor through $\mathrm{Gal}(K/\mathbf{Q})$, the trivial one and the quadratic character of K:

```
> K<i> := QuadraticField(-1);
> triv, sign := Explode(ArtinRepresentations(K));
> sign;
Artin representation of Quadratic Field with defining polynomial x^2 + 1
over the Rational Field with character ( 1, -1 )
```

An alternative way to define them is directly by their character:

```
> triv,sign:Magma;
QuadraticField(-1) !! [1,1]
QuadraticField(-1) !! [1,-1]
```

The regular representation of $\mathrm{Gal}(K/\mathbf{Q})$ is their sum:

```
> PermutationCharacter(K);
```

```
Artin representation of Quadratic Field with defining polynomial z^2 + 1
over the Rational Field with character ( 2, 0 )
> $1 eq triv+sign;
true
```

Next, let $L = K(\sqrt{-2-i})$. Then L has normal closure F with $\mathrm{Gal}(F/\mathbf{Q}) = D_4$, the dihedral group of order 8:

```
> L := ext<K|Polynomial([2+i,0,1])>;
> G := GaloisGroup(AbsoluteField(L));
> IsIsomorphic(G,DihedralGroup(4));
true
> [Dimension(A): A in ArtinRepresentations(L)];
[1, 1, 1, 1, 2 ]
```

We use `ChangeField` to lift Artin representations from $\mathrm{Gal}(K/\mathbf{Q})$ to $\mathrm{Gal}(F/\mathbf{Q})$, and check that it is still the same as an Artin representation.

```
> A := ChangeField(sign,L);
> A;
Artin representation of Number Field with defining polynomial $.1^2 + i + 2
over its ground field with character ( 1, 1, -1, 1, -1 )
> A eq sign;
true;
```

44.3 Basic Invariants

`Field(A)`

Number field K such that A factors through the Galois group of the normal closure of K.

`Degree(A)`

Degree (=dimension) of an Artin representation A.

`Group(A)`

The Galois group of the through which A factors.

`Character(A)`

Character of an Artin representation A, represented as a complex-valued character of `Group(A)`.

`Conductor(A)`

Conductor of an Artin representation A (which must be a true representation, i.e. its character is not allowed to be a generalized character). Computes all the necessary local information if Artin representations were defined with `Ramification:=false`, so the first call to this function might take some time.

`Decomposition(A)`

Decompose an Artin representation A into irreducible constituents. Returns a sequence of tuples `[...<`A_i`,`n_i`>...]` with A_i irreducible and n_i its exponent in A (nonzero but possibly negative).

`DefiningPolynomial(A)`

Returns the polynomial whose roots `Group(A)` permutes.

`Minimize(A)`

`Optimize` BoolElt *Default :* `true`

Returns A attached to the smallest number field K such that A factors through its Galois closure. If `Optimize := true`, attempts to minimize the defining polynomial of K using `OptimizedRepresentation`.

`Kernel(A)`

Smallest Galois extension K of the rationals through which A factors. Note that this field may be enormous and incomputable.

Example H44E2

We take an S_4-extension of **Q** and compute its Artin representations.

```
> R<x> := PolynomialRing(Rationals());
> K := NumberField(x^4+9*x-2);
> A := ArtinRepresentations(K);
> [Dimension(a): a in A];
[ 1, 1, 2, 3, 3 ]
```

Then we minimize the 2-dimensional one, which factors through an S_3-quotient.

```
> B := Minimize(A[3]); B;
Artin Representation of Number Field with defining polynomial x^3 + 8*x - 81
over the Rational Field with character ( 2, 0, -1 )
> Kernel(B);
Number Field with defining polynomial x^6 + 48*x^4 + 576*x^2 + 179195 over the
Rational Field
```

`IsIrreducible(A)`

Return `true` iff a given Artin representation is irreducible as a complex representation.

`IsRamified(A, p)`

Return `true` iff a given Artin representation is ramified at p.

`IsWildlyRamified(A, p)`

Return `true` iff a given Artin representation is wildly ramified at p.

`EulerFactor(A, p)`

`R` Fld *Default :* `ComplexField()`

The local polynomial (Euler factor) of an Artin representation A at the prime p. It is a polynomial with coefficients in the field R, which is complex numbers by default, and it is the inverse characteristic polynomial of (arithmetic) Frobenius at p on the inertia invariant subspace of A.

Example H44E3

Here are the invariants of Artin representations that factor through the splitting field of $x^4 - 3$, a D_4-extension of $\mathbf{Q}$.

```
> R<x> := PolynomialRing(Rationals());
> K := NumberField(x^4-3);
> A := ArtinRepresentations(K);
> Degree(Field(A[1]),Rationals());
8
> [Dimension(a): a in A];
[ 1, 1, 1, 1, 2 ]
> Character(A[5]);
( 2, -2, 0, 0, 0 )
> [Conductor(a): a in A];
[ 1, 3, 4, 12, 576 ]
> [IsRamified(a,3): a in A];
[ false, true, false, true, true ]
> [IsWildlyRamified(a,3): a in A];
[ false, false, false, false, false ]
> EulerFactor(A[5],5);
$.1^2 + 1.00000000000000000000000000000
```

`DirichletCharacter(A)`

Convert a one-dimensional Artin representation to a Dirichlet character.

`ArtinRepresentation(ch)`

`field` FldNum *Default :*

Convert a Dirichlet character `ch` to a one-dimensional Artin representation A. To avoid recomputation, the minimal field through which A factors may be supplied by the `field` parameter. This now uses class field theory (thanks to C. Fieker).

Example H44E4

An example that goes back and forth between the Dirichlet character and the Artin representation.

```
> load galpols;
> f := PolynomialWithGaloisGroup(8,46); // order 576
> K := NumberField(f); // octic field
> A := ArtinRepresentations(K);
> [Degree(a) : a in A];
[ 1, 1, 1, 1, 4, 4, 6, 6, 9, 9, 9, 9, 12 ]
> [Order(Character(Determinant(a))) : a in A];
[ 1, 2, 4, 4, 2, 2, 1, 2, 2, 1, 4, 4, 2 ]
> chi := DirichletCharacter(A[3]); // order 4
> Conductor(chi), Conductor(chi^2);
215 5
> Minimize(ArtinRepresentation(chi)); // disc = N(chi)^2*N(chi^2)
Artin representation of Number Field with defining polynomial
 x^4 - x^3 - 54*x^2 + 54*x + 551 with character ( 1, -1, I, -I )
> Factorization(Discriminant(Integers(Field($1))));
[ <5, 3>, <43, 2> ]
```

44.4 Arithmetic

`A1 + A2`

Direct sum of two Artin representations

`A1 - A2`

Direct difference of two Artin representations

`A1 * A2`

Tensor product of two Artin representations

`A1 eq A2`

Returns `true` iff the two Artin representations are equal

`A1 ne A2`

Returns `true` iff the two Artin representations are not equal

Example H44E5

For Artin representations constructed from the same number field, their arithmetic is just arithmetic of characters:

```
> P<x> := PolynomialRing(Rationals());
> K := NumberField(x^3-2);
> A := ArtinRepresentations(K: Ramification:=true);
> triv, sign, rho := Explode(A);
> triv;
Artin representation of Number Field with defining polynomial
x^3 - 2 over the Rational Field with character ( 1, 1, 1 )
and conductor 1
> rho;
Artin representation of Number Field with defining polynomial
x^3 - 2 over the Rational Field with character ( 2, 0, -1 )
and conductor 108
> triv+rho;
Artin representation of Number Field with defining polynomial
x^3 - 2 over the Rational Field with character ( 3, 1, 0 )
and conductor 108
> sign*rho eq rho;
true
```

Example H44E6

When Artin representations factor through different fields, their arithmetic involves the compositum of the fields:

```
> K1 := QuadraticField(2);
> triv1, sign1 := Explode(ArtinRepresentations(K1));
> K2 := QuadraticField(3);
> triv2, sign2 := Explode(ArtinRepresentations(K2));
> twist := sign1*sign2;
> Field(twist);
Number Field with defining polynomial $.1^4 - 10*$.1^2 + 1
over the Rational Field
> sign3 := Minimize(twist);
> sign3;
Artin representation of Number Field with defining polynomial
$.1^2 - 6 over the Rational Field with character ( 1, -1 )
> sign1*sign2*sign3 eq triv1;
true
```

44.5 Implementation Notes

The algorithms for recognizing Frobenius elements in Galois groups are described in [DD10]. They rely on the cycle type identification, Serre's trick for alternating groups and the general machinery from [DD10]. Magma is usually able to handle Galois groups of size < 10000 acting on a small number of points easily, and much larger special groups such as A_n and S_n.

44.6 Bibliography

[**DD10**] T. Dokchitser and V. Dokchitser. Identifying conjugacy classes in Galois groups. 2010.