

Handbook of Magma Functions

Part VI: Global Fields

Version 2.30-1

Sydney

October 15, 2026

Editors

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

Handbook Contributors

Eran Assaf *Geoff Bailey* *Wieb Bosma* *Gavin Brown* *Nils Bruin* *John Cannon* *Jon Carlson* *Scott Contini* *Edgar Costa* *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*

Copyright © 1994–2026 The University of Sydney. All rights reserved.

Contents

36 Number Fields **1**
- 36.1 Introduction . . . 1
- 36.2 Acknowledgement . . . 2
- 36.3 Creation Functions . . . 2
- 36.4 Structure Operations . . . 9
- 36.5 Fields with a Labelled Embedding . . . 19
- 36.6 Element Operations . . . 22
- 36.7 Class Group and Unit Group . . . 26
- 36.8 Galois Theory . . . 26
- 36.9 Solving Norm Equations . . . 27
- 36.10 Places and Divisors . . . 27
- 36.11 Number Field Database . . . 31
- 36.12 Bibliography . . . 33

37 Quadratic Fields **34**
- 37.1 Introduction . . . 34
- 37.2 Creation of Structures . . . 34
- 37.3 Operations on Structures . . . 36
- 37.4 Special Element Operations . . . 39
- 37.5 Special Functions for Ideals . . . 41
- 37.6 Bibliography . . . 42

38 Cyclotomic Fields **43**
- 38.1 Introduction . . . 43
- 38.2 Creation Functions . . . 43
- 38.3 Structure Operations . . . 45
- 38.4 Element Operations . . . 46

39 Number Fields and Orders **48**
- 39.1 Introduction . . . 48
- 39.2 Acknowledgement . . . 49
- 39.3 Creation Functions . . . 49
- 39.4 Printing . . . 64
- 39.5 Real Precision . . . 66
- 39.6 Structure Operations . . . 66
- 39.7 Element Operations . . . 83
- 39.8 Ideal Class Groups . . . 93
- 39.9 Unit Groups . . . 99
- 39.10 Diophantine Equations . . . 101
- 39.11 Ideals and Quotients . . . 107
- 39.12 Places and Divisors . . . 128
- 39.13 The Montes Algorithm . . . 131
- 39.14 Ideals in OM Representation . . . 133
- 39.15 Ideal Operations . . . 136
- 39.16 Bibliography . . . 138

40 Galois Groups and Automorphisms **141**

40.1 Automorphism Groups . . . 141
40.2 Galois Groups . . . 147
40.3 Subfields . . . 165
40.4 Galois Cohomology . . . 167
40.5 Automorphisms of Fields . . . 168
40.6 Bibliography . . . 170

41 Class Field Theory **172**
41.1 Introduction . . . 172
41.2 Creation . . . 175
41.3 Galois Module Structure . . . 185
41.4 Conversion to Number Fields . . . 186
41.5 Invariants . . . 188
41.6 Automorphisms . . . 189
41.7 Norm Equations . . . 192
41.8 Attributes . . . 194
41.9 Group Theoretic Functions . . . 200
41.10 Bibliography . . . 201

42 Étale Algebras **202**
42.1 Introduction . . . 202
42.2 Construction . . . 202
42.3 Attributes . . . 203
42.4 Homomorphisms . . . 203
42.5 Elements . . . 204
42.6 Orders of Algebras . . . 207
42.7 Ideals . . . 210
42.8 Quotients . . . 212
42.9 Over Orders . . . 214
42.10 Over Order Graph . . . 215
42.11 Trace and Norm . . . 215
42.12 Completion . . . 216
42.13 Intermediate Ideals . . . 216
42.14 Ideals of Index . . . 217
42.15 Short Element and Small Representative . . . 217
42.16 Minimal Generators . . . 218
42.17 Chinese Remainder Theorem . . . 218
42.18 Picard Group . . . 218
42.19 Factorization and Primes . . . 220
42.20 Low Cohen Macauley Type . . . 221
42.21 Weak Classes . . . 221
42.22 Weak Testing . . . 223
42.23 Ideal Class Monoid . . . 224
42.24 Complex Conjugation . . . 225
42.25 Complex Multiplication . . . 226
42.26 Totally Real and Positive . . . 227
42.27 Printing and Saving . . . 227
42.28 Bibliography . . . 228

43 Dirichlet and Hecke Characters **229**
43.1 Introduction . . . 229
43.2 Bibliography . . . 245

44 Algebraically Closed Fields **246**
44.1 Introduction . . . 246
44.2 Representation . . . 246
44.3 Creation of Structures . . . 247
44.4 Creation of Elements . . . 247
44.5 Related Structures . . . 252

44.6 Properties . . . 253
44.7 Ring Predicates and Properties . . . 253
44.8 Element Operations . . . 254
44.9 Simplification . . . 256
44.10 Absolute Field . . . 257
44.11 Bibliography . . . 262

45 Rational Function Fields **263**
45.1 Introduction . . . 263
45.2 Creation Functions . . . 263
45.3 Structure Operations . . . 265
45.4 Element Operations . . . 266
45.5 Padé-Hermite Approximants . . . 272
45.6 Bibliography . . . 279

46 Algebraic Function Fields **280**
46.1 Introduction . . . 280
46.2 Creation of Algebraic Function Fields and their Orders . . . 280
46.3 Related Structures . . . 287
46.4 General Structure Invariants . . . 291
46.5 Galois Groups . . . 295
46.6 Subfields . . . 304
46.7 Automorphism Group . . . 305
46.8 Global Function Fields . . . 311
46.9 Structure Predicates . . . 316
46.10 Homomorphisms . . . 317
46.11 Elements . . . 318
46.12 Ideals . . . 330
46.13 Places . . . 340
46.14 Divisors . . . 345
46.15 Differentials . . . 358
46.16 Weil Descent . . . 362
46.17 Function Field Database . . . 364
46.18 The Montes Algorithm . . . 365
46.19 Ideals in OM Representation . . . 368
46.20 Ideal Operations . . . 370
46.21 Divisors in OM representation . . . 373
46.22 Bibliography . . . 379

47 Class Field Theory for Global Function Fields **381**
47.1 Ray Class Groups . . . 381
47.2 Creation of Class Fields . . . 383
47.3 Properties of Class Fields . . . 386
47.4 The Ring of Witt Vectors of Finite Length . . . 387
47.5 The Ring of Twisted Polynomials . . . 390
47.6 Analytic Theory . . . 393
47.7 Related Functions . . . 398
47.8 Enumeration of Places . . . 400
47.9 Bibliography . . . 400

48 Artin Representations **402**
48.1 Overview . . . 402
48.2 Constructing Artin Representations . . . 402
48.3 Basic Invariants . . . 404
48.4 Arithmetic . . . 408
48.5 Implementation Notes . . . 409
48.6 Bibliography . . . 409

References **410**

Index **415**

Chapter 36

Number Fields

36.1 Introduction

This chapter gives an overview of number fields in Magma. Full documentation is spread across the following chapters:

Cyclotomic fields, Chapter *Cyclotomic Fields*

Quadratic fields, Chapter *Quadratic Fields*

Orders in number fields, including ideal theory, Chapter *Number Fields and Orders*

Galois Theory, Chapter *Galois Groups and Automorphisms*

Class Field Theory, Chapter *Class Field Theory*

The algorithms, functions and syntax for number fields and orders are often parallel to those for extensions of function fields of one variable.

Number fields in Magma are finite extensions of the rational field $\mathbb{Q}$, or another number field. Extensions directly over $\mathbb{Q}$ are referred to as *absolute fields*, extensions of number fields are called *relative fields*.

The `RationalField()` is *not* a number field in Magma. A trivial extension is created by `RationalsAsNumberField()`.

A number field is constructed as $K = k[t]/(f(t))$ where f is an irreducible polynomial in $k[t]$. The generator `K.1` is a root of f. A field may also be constructed as a multivariate quotient $K = k[s_1, \ldots, s_n]/(f_1(s_1), \ldots, f_n(s_n))$ for univariate f_i in $k[t]$. The generators `K.i` are roots of f_i. This construction is a shortcut for creating a tower of single extensions.

Number fields in Magma are abstract: they do not come with a distinguished embedding into an algebraically closed field. Embeddings between fields can be defined by the user, and may also be chosen (once and for all) by functions such as `IsIsomorphic` and `IsSubfield`. Crucially, it is possible to create multiple copies of the field defined by a given polynomial. Therefore every extension defined by the user creates a *new* object in Magma (unless the user requests otherwise).

The embeddings of a field into the real and complex numbers are given by the `Conjugates` of an element, or by `InfinitePlaces` of a field (and `Evaluate` to evaluate the embedding corresponding to a place). See *Places and Divisors*.

The basis of `K<w>` $= k[t]/(f(t))$ over its base field k is always the power basis $1, w, w^2, \ldots, w^{n-1}$ (where the generator `w = K.1` is a root of f). The basis of an order in an absolute field is usually in hermite form with respect to the field basis, however there are exceptions such as the `LLL` of an order. The field of fractions of an order always has the same basis as the order.

An arbitrary number field can be converted to an absolute extension of $\mathbb{Q}$ using `AbsoluteField`, i.e. this finds a primitive element over $\mathbb{Q}$. Similarly, a number field defined by multiple polynomials can be converted to a field defined by a single polynomial using *`SimpleExtension`*.

Notes about relative fields:

- Invariants such as *Degree*, *Discriminant*, *Norm*, *Trace* are always relative to the base field. They have variants of the form `Degree(K, k)` and `AbsoluteDegree`.
- Some operations and invariants are implemented only for absolute fields, for instance `ClassGroup`, `UnitGroup`, and so on.
- Conversions (eg between relative/absolute fields) may be time consuming. However the results are stored, so are only computed once. Applying the resulting maps to elements should be fast.

Warnings:

- `Discriminant(F)` for a field F returns `Discriminant(DefiningPolynomial(F))`, *not* `Discriminant(MaximalOrder(F))`.
- Some functions defined for fields are shortcuts, referring to the maximal order. For example, `UnitGroup(F)` returns the (finitely generated) unit group of `MaximalOrder(F)`, *not* the multiplicative group of F.

36.2 Acknowledgement

The number field module in Magma has mostly developed out of the Kant/Kash system (Kant-V4) [KAN97], [KAN00], developed by the group of M. Pohst at TU Berlin.

36.3 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.

36.3.1 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): RngUPolElt -> FldNum`

```
Check             : BoolElt                    Default: true
DoLinearExtension: BoolElt                     Default: false
Global            : BoolElt                    Default: false
```

Given an irreducible polynomial f of degree $n \geq 1$ over $K = \mathbb{Q}$ 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 $\mathbb{Q}$ is a field isomorphic to $\mathbb{Q}$, but regarded by Magma as a number field (while $\mathbb{Q}$ 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() -> FldNum`

`QNF() -> FldNum`

This creates a number field isomorphic to $\mathbb{Q}$. It is equivalent to `NumberField(x-1 : DoLinearExtension)`, where x is `PolynomialRing(Rationals()).1`.

The result is a field isomorphic to $\mathbb{Q}$, but regarded by Magma as a number field (while $\mathbb{Q}$ 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): [ RngUPolElt ] -> FldNum`

```
Check            : BoolElt                    Default: true
DoLinearExtension: BoolElt                    Default: false
Abs              : BoolElt                    Default: false
```

Let K be a possibly trivial algebraic extension of $\mathbb{Q}$, 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 $\mathbb{Z}$) 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< F | s1, ..., sn >* 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 >: FldNum, RngUPolElt, ..., RngUPolElt -> FldNum`

`ext<F | s>: FldNum, [RngUPolElt] -> FldNum`

`ext< Q | s1, ..., sn >: FldRat, RngUPolElt, ..., RngUPolElt -> FldNum`

`ext< Q | s >: FldRat, [RngUPolElt] -> FldNum`

```
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 $\mathbb{Q}$. 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: Creating a Number Field (ex-30ee5b)

To create the number field $\mathbb{Q}(\alpha)$, 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 $\mathbb{Q}[y]/f$:

```
> z := y^5/11;
> z;
1/11*(420*y^3 - 40000*y)
```

K can be further extended by the use of either *ext< F | s1, ..., sn >* 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);
```

(continues on next page)

(continued from previous page)

```
> 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): Rng, RngIntElt, RngElt -> FldNum`

```
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): FldNum -> FldNum, SeqEnum`

`NormalClosure(F): FldNum -> FldNum, SeqEnum`

```
Abs: BoolElt                      Default: true
Opt: BoolElt                      Default: true
```

Given a number field F, this computes 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): RngUPolElt -> FldNum

Given an irreducible polynomial f over $\mathbb{Z}$, return its splitting field.

SplittingField(L): [RngUPolElt] -> FldNum, [FldNumElt]

```
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 $\mathbb{Q}$. If in addition Opt is also true, then an optimized representation of K is computed as well.

sub< F | e_1, ..., e_n >: FldAlg, FldAlgElt, ..., FldAlgElt -> FldAlg, Map

sub< F | S >: FldAlg, SeqEnum -> FldAlg, Map

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): FldNum, FldNum -> SeqEnum

CompositeFields(F, L): FldNum, FldNum -> SeqEnum

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): FldNum, FldNum -> FldNum

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 >: RngUPol, RngUPolElt -> FldNum

```
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 $\mathbb{Q}$. 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: Normal Closure via Composite Fields (ex-180416)

To illustrate the use of CompositeFields we will use this function to compute the normal closure of $\mathbb{Q}(\alpha)$ 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): FldNum -> FldNum, Map

OptimisedRepresentation(F): FldNum -> FldNum, Map

Given a number field F with ground field $\mathbb{Q}$, 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 along with a map from F into L; otherwise F will be returned. For more details, please refer to *OptimizedRepresentation* in Chapter *Number Fields and Orders*.

Example: Optimized Representation of a Field (ex-91bae8)

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);
```

IntegralModel(F): FldNum -> FldNum, Map

Given a number field F with ground field $\mathbb{Q}$, return an isomorphic field L whose defining polynomial is monic with integer coefficients, together with an isomorphism from F to L. The transformation uses a substitution $y = dx$ for a positive integer d chosen so that the roots become algebraic integers. If the defining polynomial is already integral, returns F and the identity map.

Example: Integral Model of a Number Field (ex-67a832)

```
> R<x> := PolynomialRing(Rationals());
> K := NumberField(x^3 - 2*x + 1/2);
> L, m := IntegralModel(K);
> L;
Number Field with defining polynomial x^3 - 8*x + 4 over the Rational Field
> m(K.1);
1/2*L.1
> Inverse(m)(L.1);
2*K.1
```

36.3.2 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 Chapter *Number Fields and Orders*, 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 ([Bai96, Coh93, Poh93, PZ89] for absolute extensions and [Coh00, Fri97, Pau01b] for relative extensions).

MaximalOrder(F): FldNum -> RngOrd

IntegerRing(F): FldNum -> RngOrd

Integers(F): FldNum -> RngOrd

RingOfIntegers(F): FldNum -> RngOrd

```
Al           : MonStgElt                    Default: "Auto"
MaximalOrder: Verbose                       Default: 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 *Maximal Orders*.

36.3.3 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: FldNum, RngElt -> FldNumElt

elt< F | a >: FldNum, RngElt -> FldNumElt

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}]: FldNum, [RngElt] -> FldNumElt

elt< F | [a_0, a_1, ..., a_{m-1}] >: FldNum, SeqEnum[RngElt] -> FldNumElt

elt< F | a_0, a_1, ..., a_{m-1}>: FldNum, RngElt, ..., RngElt -> FldNumElt

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 $\alpha_i = \texttt{F.1}^i$. 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): FldNum, RngIntElt -> FldNumElt

A random element of the number field F. The maximal size of the coefficients is determined by the integer m.

Example: Creating Number Field Elements (ex-543b5e)

```
> 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,
```

(continues on next page)

(continued from previous page)

```
    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): FldNum -> FldNumElt

Identity(K): FldNum -> FldNumElt

Zero(K): FldNum -> FldNumElt

Representative(K): FldNum -> FldNumElt

36.3.4 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 >: FldNum, Rng, RngElt -> Map

hom< F -> R | h, r >: FldNum, Rng, Map, RngElt -> Map

hom< F -> R | r >: FldNum, Rng, [RngElt] -> Map

hom< F -> R | h, r >: FldNum, Rng, Map, [RngElt] -> Map

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 $G = \mathbb{Q}$) and sometimes necessary (if $G \neq \mathbb{Q}$) 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: Embedding a Quadratic Field in a Quartic Field (ex-4d9b55)

We show a way to embed the field $\mathbb{Q}(\sqrt{2})$ in $\mathbb{Q}(\sqrt{2}+\sqrt{3})$. 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
```

36.4 Structure Operations

In the lists below K always denotes a number field.

36.4.1 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): FldNum -> Cat

`Type(K): FldNum -> Cat`

`ExtendedType(K): FldNum -> Cat`

`Parent(K): FldNum -> Pow`

`AssignNames(~K, s): FldNum, [ MonStgElt ]`

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): FldNum, RngIntElt -> FldNumElt`

`K . i: FldNum, RngIntElt -> FldNumElt`

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.

36.4.2 Related Structures

Each number field has other structures related to it in various ways.

`GroundField(F): FldNum -> Fld`

`BaseField(F): FldNum -> Fld`

`CoefficientField(F): FldNum -> Fld`

`CoefficientRing(F): FldNum -> Fld`

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 $\mathbb{Q}$.

`AbsoluteField(F): FldNum -> FldNum`

Given a number field F, this returns an isomorphic number field L defined as an absolute extension (i.e. over $\mathbb{Q}$). (For algorithm, see [Tra76])

`SimpleExtension(F): FldNum -> FldNum`

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): FldNum, FldNum -> FldNum`

`RelativeField(Q, L): FldRat, FldNum -> FldNum`

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.

`Components(F): FldNum -> [FldNum]`

Given a number field F return the sequence of number fields each defined by a defining polynomial of F.

Example: Compositum of Two Number Fields `(ex-e3e873)`

It is often desirable to build up a number field by adjoining several algebraic numbers to $\mathbb{Q}$. The following function returns a number field that is the composite field of two given number fields K and L, provided that $K \cap L = \mathbb{Q}$; if K and L have a common subfield larger than $\mathbb{Q}$ 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 $\mathbb{Q}(\sqrt{2}, \sqrt{3}, \sqrt{5})$, 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): FldNum -> RngRat

PrimeField(F): FldNum -> RngRat

Centre(F): FldNum -> FldNum

Embed(F, L, a): FldNum, FldNum, FldNumElt

```
Overwrite: BoolElt                    Default: false
```

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.

If the addition of this embedding causes an inconsistency with currently known embeddings then the embedding will not be added unless `Overwrite` is set to `true`.

Embed(F, L, a): FldNum, FldNum, [FldNumElt]

```
Overwrite: BoolElt                    Default: false
```

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.

If the addition of this embedding causes an inconsistency with currently known embeddings then the embedding will not be added unless `Overwrite` is set to `true`.

EmbeddingMap(F, L): FldNum, FldNum -> Map

Returns the embedding map of the number field F in L if an embedding is known.

Example: Embedding One Number Field in Another (ex-909e79)

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] : Overwrite := true);
> L!l.1;
1/36*(-b^4 - 18*b)
```

MinkowskiSpace(F): FldNum -> Lat, Map

Precision: RngIntElt	Default: 167

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): FldNum, RngOrdIdl -> FldLoc, Map

comp<K|P>: FldNum, RngOrdIdl -> FldLoc, Map

Precision: RngIntElt	Default: 20

For an absolute extension K of $\mathbb{Q}$, 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 default precision **Precision**.

The returned map is the canonical injection into the completion. It allows pointwise inverse operations.

Completion(K, P): FldNum, PlcNumElt -> FldLoc, Map

Precision: RngIntElt	Default: 20

For an absolute extension K over $\mathbb{Q}$ and a (finite) place P, compute the completion at P. The precision and the map are as described for *Completion*.

36.4.3 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): FldNum, Fld -> AlgAss, Map

Algebra(K, J, S): FldNum, Fld, [FldNumElt] -> AlgAss, Map

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): FldNum, Fld -> ModTupFld, Map

KSpace(K, J): FldNum, Fld -> ModTupFld, Map

VectorSpace(K, J, S): FldNum, Fld, [FldNumElt] -> ModTupFld, Map

KSpace(K, J, S): FldNum, Fld, [FldNumElt] -> ModTupFld, Map

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: Minimal Polynomial over a Subfield (ex-3f6dca)

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;
```

(continues on next page)

(continued from previous page)

```
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
```

36.4.4 Invariants

Some information describing a number field can be retrieved.

Characteristic(F): FldNum -> RngIntElt

Degree(F): FldNum -> RngIntElt

Given a number field F, return the degree $[F : G]$ of F over its ground field G.

AbsoluteDegree(F): FldNum -> RngIntElt

Given a number field F, return the absolute degree of F over $\mathbb{Q}$.

Discriminant(F): FldNum -> RngIntElt

Discriminant(F): FldNum -> RngOrdIdl

Given an extension F of $\mathbb{Q}$, 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): FldNum -> FldRatElt

Given a number field K, return the absolute value of the discriminant of K regarded as an extension of $\mathbb{Q}$.

Regulator(K): FldNum -> FldComElt

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): FldNum -> FldComElt

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): FldAlg -> RngIntElt, RngIntElt

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): FldNum -> RngIntElt

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): FldNum -> RngUPolElt

DefiningPolynomial(F): FldNum -> [RngUPolElt]

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): FldNum, RngIntElt -> [FldComElt]

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: Zeros and Defining Polynomial (ex-e63cd9)

The information provided by **Zeros** and **DefiningPolynomial** is illustrated below.

```
> L := NumberField(x^6+108);
> DefiningPolynomial(L);
x^6 + 108
> Zeros(L, 30);
[ 1.889881574842309747150815910899999999994 +
1.091123635971721403560072614199999999977*i,
1.889881574842309747150815910899999999994 -
1.091123635971721403560072614199999999977*i,  0.E-29 +
2.182247271943442807120145228399999999955*i,  0.E-29 -
2.182247271943442807120145228399999999955*i,
-1.889881574842309747150815910899999999994 +
1.091123635971721403560072614199999999977*i,
-1.889881574842309747150815910899999999994 -
1.091123635971721403560072614199999999977*i ]
```

(continues on next page)

(continued from previous page)

```
> 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 ]
```

36.4.5 Basis Representation

The basis of a number field can be expressed using elements from any compatible ring.

Basis(F): FldNum -> [FldNumElt]

Basis(F, R): FldNum, Rng -> [RngElt]

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): FldNum -> [FldNumElt]

IntegralBasis(F, R): FldNum, Rng -> [RngElt]

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: Basis over Different Rings (ex-49b3f2)

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],
```

(continues on next page)

(continued from previous page)

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

AbsoluteBasis(K): FldNum -> [FldNumElt]

Returns an absolute basis for the number field K, i.e. a basis for K as a $\mathbb{Q}$ vector space. The basis will consist of the products of the basis elements of the intermediate fields. The expansion is done depth-first.

Example: Basis and Integral Basis (ex-0dc237)

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)
]
[
    0.1,
    0.2,
    0.3,
    0.4
]
> Basis(O, K);
[
    1,
    1/2*y,
    1/40*(y^2 + 10*y),
    1/800*(y^3 + 180*y + 400)
]
```

36.4.6 Ring Predicates

Number fields can be tested for having several properties that may hold for general rings.

F eq L: FldNum, FldNum -> BoolElt

F eq L: FldRat, FldNum -> BoolElt

F eq L: FldNum, FldRat -> BoolElt

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): Rng -> BoolElt

IsUnitary(R): Rng -> BoolElt

IsFinite(R): Rng -> BoolElt

IsOrdered(R): Rng -> BoolElt

IsField(R): Rng -> BoolElt

IsNumberField(R): . -> BoolElt

IsAlgebraicField(R): . -> BoolElt

IsEuclideanDomain(F): FldNum -> BoolElt

This is not a check for euclidean number fields. This function will always return **true**, as all number fields are euclidean domains.

IsSimple(F): FldNum -> BoolElt

Checks if the number field F is defined as a simple extension over the base ring.

IsPID(F): FldNum -> BoolElt

IsUFD(F): FldNum -> BoolElt

IsPrincipalIdealRing(F): FldNum -> BoolElt

Always **true** for number fields.

IsDomain(R): FldNum -> BoolElt

F **ne** L: FldNum, FldNum -> BoolElt

K **subset** L: FldNum, FldNum -> BoolElt

K **subset** L: FldRat, FldNum -> BoolElt

HasComplexConjugate(K): FldNum -> BoolElt, Map

This function returns **true** if there is an automorphism in the number field K that acts like complex conjugation.

ComplexConjugate(x): FldNumElt -> FldNumElt

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.

36.4.7 Field Predicates

Here all the predicates that are specific to number fields are listed.

IsIsomorphic(F, L): FldNum, FldNum -> BoolElt, Map

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): FldNum, FldNum -> BoolElt, Map

IsSubfield(F, L): FldRat, FldNum -> BoolElt, Map

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): FldNum -> BoolElt

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): FldNum -> BoolElt

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): FldNum -> BoolElt

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): FldNum -> BoolElt

Returns **true** iff the number field K is a constructed as an absolute extension of $\mathbb{Q}$.

36.5 Fields with a Labelled Embedding

A Number Field has several diffenent embeddings into the complex numbers. An embedded Number Field is a Number Field such that one of these embeddings is labelled. These fields form a category named `FldNumEmb`.

36.5.1 Creation Functions

Functions are provided to create fields of the special type `FldNumEmb`.

`EmbeddedNumberField(L, c): FldNum, FldComElt -> FldNumEmb`

`EmbeddedNumberField(L, c): FldNum, FldReElt -> FldNumEmb`

Turns L into a Number Field with a labelled embedding that extends the labelled embedding of its base field and maps its generator to c. It reports an error when no embedding fits c, as when c is too far from every root. It also reports an error when c is too coarse to distinguish matching embeddings. Use `HasEmbeddedNumberField` to test without an error.

`HasEmbeddedNumberField(L, c): FldNum, FldComElt -> BoolElt, FldNumEmb`

`HasEmbeddedNumberField(L, c): FldNum, FldReElt -> BoolElt, FldNumEmb`

Returns `true` if exactly one embedding of L extends the labelled embedding of its base field and maps its generator to c. Returns `false` when no embedding fits c or when c is too coarse to distinguish matching embeddings. When the answer is `true`, the second value is L as a Number Field with that labelled embedding.

`EmbeddedNumberField(f, r): RngUPolElt, FldReElt -> FldNumEmb`

`EmbeddedNumberField(f, r): RngUPolElt, FldComElt -> FldNumEmb`

Constructs the Number Field defined by f with a labelled embedding. The embedding extends that of the coefficient field and maps the generator to r. It reports an error when no embedding fits r or when r is too coarse to distinguish matching embeddings. Use `HasEmbeddedNumberField` to test without an error.

`HasEmbeddedNumberField(f, r): RngUPolElt, FldReElt -> BoolElt, FldNumEmb`

`HasEmbeddedNumberField(f, r): RngUPolElt, FldComElt -> BoolElt, FldNumEmb`

Returns `true` if the Number Field defined by f has exactly one matching embedding. That embedding extends the labelled embedding of its coefficient field and maps its generator to r. Returns `false` when no embedding fits r or when r is too coarse to distinguish matching embeddings. When the answer is `true`, the second value is that Number Field with the labelled embedding.

`EmbeddedNumberField(f, i): RngUPolElt, RngIntElt -> FldNumEmb`

Constructs a Number Field with defining polynomial f and labelled embedding with number i.

`EmbeddedNumberField(L, i): FldNum, RngIntElt -> FldNumEmb`

Constructs a Number Field with labelled embedding with number i and the same defining polynomial as L.

`EmbeddedSplittingField(f): RngUPolElt -> FldNumEmb, SeqEnum`

Constructs the splitting field of f as number field with labelled embedding. The second return value are the roots of f in the splitting field.

`Subfield(L, K): FldNumEmb, FldNum -> FldNumEmb`

Turns the subfield K of the Number Field L with labelled embedding into a number field with labelled embedding.

36.5.2 Composition and Intersection

Number fields with a fixed embedding into the complex numbers have a unique composition and intersection as subfields of the complex numbers. The following functions are availabe to work in the subfield lattice.

`Composite(K, L): FldNumEmb, FldNumEmb -> FldNumEmb`

The unique composite of K and L as a number field with labelled embedding.

`Intersection(K, L): FldNumEmb, FldNumEmb -> FldNumEmb`

`K meet L: FldNumEmb, FldNumEmb -> FldNumEmb`

The unique intersection of K and L as a number field with labelled embedding.

`IsSubfieldEmb(K, L): FldNumEmb, FldNumEmb -> BoolElt`

Returns `true` if the image of K in the complex numbers is a subfield of the image of L and `false` otherwise.

36.5.3 Embedding and Reconstruction

The following functions are available to work with elements of a number field with a labelled embedding.

`ComplexImage(x): FldNumElt -> FldComElt`

Precision: RngIntElt	Default: 30

The image of x in the complex numbers.

`Embedding(K): FldNumEmb -> UserProgram`

Precision: RngIntElt	Default: 30

The labelled embeding of K into the complex numbers as a user program.

`Reconstruction(L, x): FldNumEmb, FldComElt -> BoolElt, FldNumElt`

`Reconstruction(L, x): FldNumEmb, FldReElt -> BoolElt, FldNumElt`

UseLLLOrder: BoolElt	Default: false

Attempts to reconstruct x as an element in L. Returns `true` if successful and `false` otherwise. If the option `UseLLLOrder` is set, and LLL basis of the maximal order will be used.

Example: Creating an Embedded Number Field `(ex-83530f)`

The approximation 2.0 is too far from $7^{1/3}$ to identify an embedding. The approximation 1.913 is close enough.

The polynomial $x^2 - 2x + 1 - 2/10^{12}$ has roots $1 \pm \sqrt{2}/10^6$. An approximation with five significant digits does not distinguish them. An approximation with 20 digits does:

```
> _<x> := PolynomialRing(Integers());
> K := NumberField(x^3-7);
> HasEmbeddedNumberField(K,2.0);
false
> HasEmbeddedNumberField(K,1.913);
true Number Field with defining polynomial x^3 - 7 over the Rational
Field
> f := ChangeRing(x^2 - 2*x + 1, Rationals()) - 2/10^12;
> HasEmbeddedNumberField(f, RealField(5)!1.0000);
false
> R := RealField(20);
> ok, M := HasEmbeddedNumberField(f, R!1.0000014142135623731);
> ok;
true
> ComplexImage(M.1);
1.0000014142135623731309504880169
```

(continues on next page)

(continued from previous page)

```
> L := EmbeddedNumberField(K,7^(1/3));
> ComplexImage(L.1 + L.1^2);
5.57223689279536061843719014967
> Reconstruction(L,1/2 + 7^(1/3) + 5 * 7^(2/3));
true 1/2*(10*L.1^2 + 2*L.1 + 1)
> Reconstruction(L,Sqrt(-1));
false
```

Intersection and composition can be used as follows:

```
> _<x> := PolynomialRing(Integers());
> K := EmbeddedNumberField(x^6-3,3^(1/6));
> L := EmbeddedNumberField(x^9-3,3^(1/9));
> K meet L;
Number Field with defining polynomial $.1^3 + 9*$.1^2 + 27*$.1 + 18
over the Rational Field
> C := Composite(K,L);
> Degree(C);
18
> IsSubfieldEmb(K,C) and IsSubfieldEmb(L,C);
true
```

Here is an example to work with towers of fields:

```
> _<x> := PolynomialRing(Integers());
> K := EmbeddedNumberField(x^2-2,Sqrt(2));
> L := EmbeddedNumberField(Polynomial([-K.1,0,1]),2^(1/4));
> ComplexImage(L.1);
1.189207115002721066717499970 56
> ComplexImage(L!(K.1));
1.414213562373095048801688724 21
```

The labelled embeddings determine the composite. Let w be the cube root of 2 with positive imaginary part. Label $K = \mathbb{Q}(2^{1/3})$ at the complex conjugate of w, and label $L = \mathbb{Q}(\sqrt{2})(2^{1/3})$ at w. Their composite contains both w and its conjugate. It therefore contains ζ_3 and is $\mathbb{Q}(\sqrt{2}, 2^{1/3}, \zeta_3)$, of degree 12. The displayed images of the generators verify that the composite retains both labelled embeddings:

```
> _<x> := PolynomialRing(Integers());
> CC<i> := ComplexField(30);
> w := 2^(1/3) * (-1 + Sqrt(CC!-3))/2;
> w;
-0.629960524947436582383605303639 + 1.09112363597172140356007261419*i
> K := EmbeddedNumberField(x^3-2, ComplexConjugate(w));
> S := EmbeddedNumberField(x^2-2, Sqrt(2));
> _<y> := PolynomialRing(S);
> L := EmbeddedNumberField(y^3-2, w);
> C := Composite(K,L);
> AbsoluteDegree(C);
12
> ComplexImage(C!K.1);
-0.629960524947436582383605303639 - 1.09112363597172140356007261419*i
> ComplexImage(C!L.1);
-0.629960524947436582383605303639 + 1.09112363597172140356007261419*i
```

36.6 Element Operations

36.6.1 Parent and Category

`Parent(a): FldNumElt -> FldNum`

`Category(a): FldNumElt -> Cat`

`Type(a): FldNumElt -> Cat`

`ExtendedType(a): FldNumElt -> Cat`

36.6.2 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: FldNumElt -> FldNumElt`

`- a: FldNumElt -> FldNumElt`

`a + b: FldNumElt, FldNumElt -> FldNumElt`

`a - b: FldNumElt, FldNumElt -> FldNumElt`

`a * b: FldNumElt, FldNumElt -> FldNumElt`

`a / b: FldNumElt, FldNumElt -> FldNumElt`

`a ^ k: FldNumElt, RngIntElt -> FldNumElt`

`Sqrt(a): FldNumElt -> FldNumElt`

`SquareRoot(a): FldNumElt -> FldNumElt`

Returns the square root of the number field element a if it exists in the field containing a.

`Root(a, n): FldNumElt, RngIntElt -> FldNumElt`

Returns the n-th root of the number field element a if it exists in the field containing a.

`IsPower(a, k): FldNumElt, RngIntElt -> BoolElt, FldNumElt`

`IsSquare(a): FldNumElt -> BoolElt, FldNumElt`

Return `true` if the number field element a is a kth power, (respectively square) and the root if so.

`Denominator(a): FldNumElt -> RngIntElt`

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): FldNumElt -> RngIntElt`

Returns the numerator of the number field element a, that is the element multiplied by its denominator.

`Qround(E, M): FldNumElt, RngIntElt -> FldNumElt`

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.

36.6.3 Equality and Membership

Elements may also be tested for whether they lie in an ideal of an order. See Section *Predicates on Ideals*.

`a eq b: FldNumElt, FldNumElt -> BoolElt`

`a ne b: FldNumElt, FldNumElt -> BoolElt`

`a in F: FldNumElt, FldNum -> BoolElt`

36.6.4 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): FldNumElt -> BoolElt, RngIntElt`

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): FldNumElt -> BoolElt`

Returns `true` if the element a of the number field F generates F over its coefficient field.

`IsTotallyPositive(a): FldNumElt -> BoolElt`

Returns `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): FldNumElt -> BoolElt`

`IsOne(a): FldNumElt -> BoolElt`

`IsMinusOne(a): FldNumElt -> BoolElt`

`IsUnit(a): FldNumElt -> BoolElt`

`IsNilpotent(a): FldNumElt -> BoolElt`

`IsIdempotent(a): FldNumElt -> BoolElt`

`IsZeroDivisor(a): FldNumElt -> BoolElt`

`IsRegular(a): FldNumElt -> BoolElt`

`IsIrreducible(a): FldNumElt -> BoolElt`

`IsPrime(a): FldNumElt -> BoolElt`

36.6.5 Field Generators

`K . 1: FldNum -> FldNumElt`

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): FldNum -> FldNumElt`

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): FldNum -> {FldNumElt}`

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): FldNum -> {FldNumElt}`

A set of generators of the number field K over $\mathbb{Q}$.

GeneratorsSequence(K): FldNum -> [FldNumElt]

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): FldNum -> [FldNumElt]

A sequence of generators of the number field K over $\mathbb{Q}$.

Generators(K, k): FldAlg, FldAlg -> [FldAlgElt]

Generators(K, Q): FldAlg, FldRat -> [FldAlgElt]

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.

36.6.6 Real and Complex Embeddings

See Chapter *Number Fields and Orders* (on number fields and orders).

36.6.7 Heights

See Chapter *Number Fields and Orders* (on number fields and orders).

36.6.8 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 $\mathbb{Z}$ or $\mathbb{Q}$.

Norm(a): FldNumElt -> FldNumElt

Norm(a, R): FldNumElt, Rng -> RngElt

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): FldNumElt -> FldRatElt

NormAbs(a): FldNumElt -> FldRatElt

The absolute norm $\mathrm{N_{L/\mathbb{Q}}(a)}$ over $\mathbb{Q}$ of the element a of the number field L.

Trace(a): FldNumElt -> FldNumElt

Trace(a): FldNumElt -> FldRatElt

Trace(a, R): FldNumElt, Rng -> RngElt

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): FldNumElt -> FldRatElt

TraceAbs(a): FldNumElt -> FldRatElt

The absolute trace $\mathrm{Tr}_{L/\mathbb{Q}}(a)$ over $\mathbb{Q}$ of the element a of the number field L.

CharacteristicPolynomial(a): FldNumElt -> RngUPolElt

CharacteristicPolynomial(a, R): FldNumElt, Rng -> RngUPolElt

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): FldNumElt -> RngUPolElt

Given an element a from a number field, this function returns the characteristic polynomial of a over $\mathbb{Q}$.

MinimalPolynomial(a): FldNumElt -> RngUPolElt

MinimalPolynomial(a, R): FldNumElt, Rng -> RngUPolElt

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): FldNumElt -> RngUPolElt

Given an element a from a number field, this function returns the minimal polynomial of the element as a polynomial over $\mathbb{Q}$.

RepresentationMatrix(a): FldNumElt -> NumMatElt

RepresentationMatrix(a, R): FldNumElt, Rng -> NumMatElt

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): FldNumElt -> NumMatElt

Return the representation matrix of the number field element a relative to the $\mathbb{Q}$-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 $L_0 := \mathbb{Q}$. Then the representation matrix is computed with respect to the $\mathbb{Q}$-basis $(\prod_j \omega_{i_j,j})_{i\in I}$ consisting of products of basis elements of the different levels.

Example: Norm, Trace, and Minimal Polynomial (ex-441d85)

We create the norm, trace, minimal polynomial, and representation matrix of the element $\alpha/2$ in the quartic field $\mathbb{Q}(\alpha)$.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^4-420*x^2+40000);
> z := y/2;
> Norm(z), Trace(z);
2500 0
> MinimalPolynomial(z);
$.1^4 - 105*$.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
```

36.6.9 Other Functions

Elements can be represented by sequences and have a product representation.

ElementToSequence(a): FldNumElt -> [FldNumElt]

Eltseq(a): FldNumElt -> [FldNumElt]

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): FldNumElt, FldNum -> [RngElt]

`Eltseq(E, k): FldNumElt, Rng -> [RngElt]`

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): FldNumElt -> [ FldRatElt]`

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]: FldNumElt, RngIntElt -> FldRatElt`

`a[i]: FldNumElt, RngIntElt -> FldNumElt`

The coefficient of the ith basis element in the number field element a.

`ProductRepresentation(a): FldNumElt -> [ FldNumElt ], [ RngIntElt ]`

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): [ FldNumElt ], [ RngIntElt ] -> FldNumElt`

`PowerProduct(P, E): [FldNumElt], [RngIntElt] -> FldNumElt`

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.

36.7 Class Group and Unit Group

The `ClassGroup` routine and related functions are described in Section *Ideal Class Groups*. The routines for computing units in maximal orders of number fields are described in Section *Unit Groups*.

Example: Class Group and Unit Group `(ex-c8082f)`

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
```

36.8 Galois Theory

`GaloisGroup(K): FldNum -> GrpPerm, [RngElt], GaloisData`

`Subfields(K): FldNum -> [<FldNum, Map>]`

`AutomorphismGroup(K): FldNum -> GrpPerm, [Map], Map`

See Sections *Galois Groups*, *Subfields*, and *Automorphism Groups*.

36.9 Solving Norm Equations

Magma has routines for solving norm equations, Thue equations, index form equations, and unit equations. These are documented in Section *Diophantine Equations*.

Example: Solving a Relative Norm Equation (ex-2b3577)

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
```

36.10 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 $\mathbb{R}$ or with pairs of embeddings into $\mathbb{C}$.

The group of divisors is formally the free group generated by the finite places and the $\mathbb{R}$-vectorspace generated by the infinite ones. Divisors are of type `DivNumElt`. Places have formal parent of type `PlcNum`, while divisors belong to `DivNum`.

36.10.1 Creation of Structures

`Places(K): FldNum -> PlcNum`

`DivisorGroup(K): FldNum -> DivNum`

The set of places of the number field K and the group of divisors of K respectively.

36.10.2 Operations on Structures

`d1 eq d2: DivNum, DivNum -> BoolElt`

`p1 eq p2: PlcNum, PlcNum -> BoolElt`

`NumberField(P): PlcNum -> FldNum`

`NumberField(D): DivNum -> FldNum`

The number field for which P is the set of places or D is the group of divisors.

36.10.3 Creation of Elements

`Place(I): RngOrdIdl -> PlcNumElt`

The place corresponding to prime ideal I.

`Decomposition(K, p): FldNum, RngIntElt -> SeqEnum`

`Decomposition(K, I): FldNum, Infty -> SeqEnum`

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): FldNum, PlcNumElt -> SeqEnum`

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): Map[FldRat, FldNum], RngIntElt -> SeqEnum[<PlcNumElt, RngIntElt>]`

`Decomposition(m, p): Map[FldNum, FldNum], PlcNumElt -> SeqEnum[<PlcNumElt, RngIntElt>]`

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): FldNum -> SeqEnum`

`InfinitePlaces(K): FldRat -> SeqEnum`

`InfinitePlaces(K): RngInt -> SeqEnum`

A sequence containing all the infinite places of the number field K is returned.

`Divisor(pl): PlcNumElt -> DivNumElt`

The divisor $1 * pl$ for a place pl.

`Divisor(I): RngOrdFracIdl -> DivNumElt`

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): FldNumElt -> DivNumElt`

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): FldRat -> [PlcNumElt]`

`RealPlaces(K): FldNum -> [PlcNumElt]`

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.

36.10.4 Arithmetic with Places and Divisors

Divisors and places can be added, negated, subtracted and multiplied and divided by integers.

`d1 + d2: DivNumElt, DivNumElt -> DivNumElt`

`p + d: PlcNumElt, DivNumElt -> DivNumElt`

`d + p: DivNumElt, PlcNumElt -> DivNumElt`

`p1 + p2: PlcNumElt, PlcNumElt -> DivNumElt`

`- p: PlcNumElt -> DivNumElt`

`- d: DivNumElt -> DivNumElt`

`d - p: DivNumElt, PlcNumElt -> DivNumElt`

`p - d: PlcNumElt, DivNumElt -> DivNumElt`

`d1 - d2: DivNumElt, DivNumElt -> DivNumElt`

`p1 - p2: PlcNumElt, PlcNumElt -> DivNumElt`

`p * k: PlcNumElt, RngIntElt -> DivNumElt`

`d * k: DivNumElt, RngIntElt -> DivNumElt`

`p div k: PlcNumElt, RngIntElt -> DivNumElt`

`d div k: DivNumElt, RngIntElt -> DivNumElt`

36.10.5 Other Functions for Places and Divisors

`Valuation(a, p): FldNumElt, PlcNumElt -> RngElt`
`Valuation(a, p): RngElt, PlcNumElt -> RngElt`

The valuation of the element a of a number field at the place p.

`Valuation(I, p): RngOrdFracIdl, PlcNumElt -> RngElt`

The valuation of the ideal I at the finite place p.

`Support(D): DivNumElt -> SeqEnum, SeqEnum`

The support of the divisor D as a sequence of places and a sequence of the corresponding exponents.

`Ideal(D): DivNumElt -> RngOrdIdl`
`Ideal(D): PlcNumElt -> RngOrdIdl`

The ideal corresponding to the finite part of the divisor D.

`Evaluate(x, p): FldNumElt, PlcNumElt -> RngElt`
`Evaluate(x, p): FldRatElt, PlcNumElt -> RngElt`
`Evaluate(x, p): RngIntElt, Infty -> RngElt`

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): FldNumElt -> []`

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): FldNumElt -> []`

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): PlcNumElt -> BoolElt`

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): PlcNumElt -> BoolElt`

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): PlcNumElt -> BoolElt`

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): PlcNumElt -> BoolElt, RngIntElt`

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): PlcNumElt, PlcNumElt -> BoolElt`

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 divides, 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): PlcNumElt -> RngIntElt`

`Degree(P): PlcNumElt -> RngIntElt`

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): DivNumElt -> RngElt`

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): PlcNumElt -> FldNum`

`NumberField(D): DivNumElt -> FldNum`

For a place P or divisor D of a number field, return the underlying number field.

`ResidueClassField(P): PlcNumElt -> Fld`

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): PlcNumElt -> FldNumElt`

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): PlcNumElt -> RngIntElt`

The degree of the completion at the place P, i.e. the product of the inertia degree times the ramification index.

`RamificationIndex(P): PlcNumElt -> RngIntElt`

The ramification index of the place P. For infinite real places this is 1 and 2 for complex places.

`DecompositionGroup(P): PlcNumElt -> GrpPerm`

For a place P of a normal number field, return the decomposition group as a subgroup of the (abstract) automorphism group.

36.11 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< D | dmin, dmax : parameters>* constructor: The discriminant; the signature; the Galois group; the class number; and the class group.

36.11.1 Creation

NumberFieldDatabase(d): RngIntElt -> DB

Returns a database object for the number fields of degree d.

sub< D | dmin, dmax : parameters>: DB, RngIntElt, RngIntElt -> DB

sub< D | dabs : parameters>: DB, RngIntElt -> DB

sub< D | : parameters>: DB -> DB

```
Signature            : [ RngIntElt ]                          Default:
Signatures           : [ [RngIntElt] ]                        Default:
GaloisGroup          : RngIntElt or GrpPerm                   Default:
ClassNumber          : RngIntElt                              Default:
ClassNumberLowerBound: RngIntElt                              Default: 1
ClassNumberUpperBound: RngIntElt                              Default:␣
↪Infinity()
ClassNumberBounds    : [ RngIntElt ]                          Default: [ 1,␣
↪Infinity() ]
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 super-groups 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.

36.11.2 Access

`Degree(D): DB -> RngIntElt`

Returns the degree of the number fields stored in the database.

`DiscriminantRange(D): DB -> RngIntElt, RngIntElt`

Returns the smallest and largest discriminants of the number fields stored in the database.

`# D: DB -> RngIntElt`

`NumberOfFields(D): DB -> RngIntElt`

Returns how many number fields are stored in the database.

`NumberOfFields(D, d): DB, RngIntElt -> RngIntElt`

Returns how many number fields of discriminant d are stored in the database.

`NumberFields(D): DB -> [ FldNum ]`

Returns the sequence of number fields stored in the database.

`NumberFields(D, d): DB, RngIntElt -> [ FldNum ]`

Returns the sequence of number fields of discriminant d stored in the database.

Example: Basic Access to the Number Field Database (ex-461fd7)

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: Quintic Fields with a Given Signature (ex-07449b)

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
```

36.12 Bibliography

- [Bai96] Georg Baier. Zum Round 4 Algorithmus. Master's thesis, Technische Universität Berlin, 1996. http://www.math.tu-berlin.de/∼kant/publications/diplom/baier.ps.gz.
- [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.
- [Fri97] Carsten Friedrichs. Berechnung relativer Ganzheitsbasen mit dem Round-2-Algorithmus. Master's thesis, Technische Universität Berlin, 1997. http://www.math.tu-berlin.de/∼kant/publications/diplom/friedrichs.ps.gz.
- [Pau01b] 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.
- [Tra76] Barry M. Trager. Algebraic factoring and rational function integration. In R.D. Jenks, editor, *Proc. SYMSAC '76*, 196–208. ACM press, 1976.
- [KAN97] KANT Group. KANT V4. *J. Symbolic Comp.*, 24(3–4):267–383, 1997.
- [KAN00] KANT Group. The Number Theory Package KANT/KASH. http://www.math.tu-berlin.de/∼kant, 2000.

Chapter 37

Quadratic Fields

37.1 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 *Number Fields*.

The categories involved are `FldQuad` for fields, `RngQuad` for their orders and `FldQuadElt` and `RngQuadElt` for their elements.

37.1.1 Representation

For every squarefree integer d (not 0 or 1) there is a unique quadratic field $\mathbb{Q}(\sqrt{d})$; for any integer k we have the field $\mathbb{Q}(\sqrt{k^2 d}) \cong \mathbb{Q}(\sqrt{d})$. Given any integer m, the function *QuadraticField* will create a structure corresponding to the quadratic field $\mathbb{Q}(\sqrt{d})$, 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 $\mathbb{Q}(\sqrt{d})$ has been created before a reference on it will be returned: two fields with the same d are the same. The discriminant D of $\mathbb{Q}(\sqrt{d})$ will be $D = d$ if $d \equiv 1 \bmod 4$ and $D = 4d$ if $d \equiv 2, 3 \bmod 4$. Elements of $\mathbb{Q}(\sqrt{d})$ 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 $F = \mathbb{Q}(\sqrt{d})$ will be $O_F = \mathbb{Z} + \epsilon_d\mathbb{Z}$, 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 $F = \mathbb{Q}(\sqrt{d})$, 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 $E_F = \mathbb{Z} + \sqrt{d}\mathbb{Z}$. Suborders of conductor f can be formed which will contain elements of the form $x + yf\sqrt{d}$ for any integers x and y.

37.2 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 *Creation of Elements*.

`QuadraticField(m): RngIntElt -> FldQuad`

Given an integer m that is not a square, create the field $\mathbb{Q}(\sqrt{d})$, 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): FldQuad -> RngQuad`

Creation of the order $\mathbb{Z}[\sqrt{d}]$ in the quadratic field $F = \mathbb{Q}(\sqrt{d})$, with d squarefree.

`MaximalOrder(F): FldQuad -> RngQuad`

`IntegerRing(F): FldQuad -> RngQuad`

`RingOfIntegers(F): FldQuad -> RngQuad`

Given a quadratic field $F = \mathbb{Q}(\sqrt{d})$, with d squarefree, create its maximal order. This order is $\mathbb{Z}[\sqrt{d}]$ if $d \equiv 2, 3 \bmod 4$ and $\mathbb{Z}[\frac{1+\sqrt{d}}{2}]$ if $d \equiv 1 \bmod 4$.

`NumberField(O): RngQuad -> FldQuad`

Given a quadratic order, this returns the quadratic field of which it is an order.

`sub< O | f >: RngQuad, RngIntElt`

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): FldNum -> BoolElt, FldQuad`

`IsQuadratic(O): RngOrd -> BoolElt, RngQuad`

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: Creating a Quadratic Field and Order (ex-a2440a)

We create the quadratic field $\mathbb{Q}(\sqrt{5})$ 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: Embedding a Quadratic Field in a Cyclotomic Field (ex-6098de)

We define an injection $\phi\colon \mathbb{Q}(\sqrt{5}) \to \mathbb{Q}(\zeta_5)$. First a square root of 5 is identified in $\mathbb{Q}(\zeta_5)$.

```
> Q<w> := QuadraticField(5);
> F<z> := CyclotomicField(5);
> C<c> := PolynomialRing(F);
> Factorization(c^2-5);
[
```

(continues on next page)

(continued from previous page)

```
   <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
```

37.3 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]): FldQuad, [MonStgElt]

AssignNames(~O, [s]): RngQuad, [MonStgElt]

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 $\mathbb{Q}(\sqrt{d})$ 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 ~ in the call to this function.

Name(F, 1): FldQuad, RngIntElt -> FldQuadElt

Name(O, 1): RngQuad, RngIntElt -> RngQuadElt

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): FldQuad -> FldQuadElt

FundamentalUnit(O): RngQuad -> RngQuadElt

A generator for the unit group of the order O or the maximal order of the quadratic field K.

Discriminant(K): FldQuad -> RngIntElt

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): FldQuad -> RngIntElt, [RngIntElt]

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): RngQuad -> RngIntElt

The conductor of the order O, which equals the index of O in the maximal order.

37.3.1 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(Q: parameters): QuadBin → GrpAb, Map` in Chapter Binary Quadratic Forms for details.

ClassGroup(K): FldQuad -> GrpAb, Map

ClassGroup(O): RngQuad -> GrpAb, Map

```
FactorBasisBound: FldReElt                    Default: 0.1
ProofBound      : FldReElt                    Default: 6
ExtraRelations  : RngIntElt                   Default: 1
Al              : MonStgElt                   Default: "Automatic"
ClassGroupSieve : Verbose                     Default: 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(Q: parameters): QuadBin → GrpAb, Map` in Chapter Binary Quadratic Forms.

`ClassNumber(K): FldQuad -> RngIntElt`

`ClassNumber(O): RngQuad -> RngIntElt`

```
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): RngQuad -> GrpAb, Map`

`PicardNumber(O): RngQuad -> RngIntElt`

```
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: Class Groups by Sieving `(ex-a05688)`

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
    14876959870232744646703839088 8*$.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): FldQuad -> GrpAb, Map

QuadraticClassGroupTwoPart(O): RngQuad -> GrpAb, Map

QuadraticClassGroupTwoPart(d): RngIntElt -> GrpAb, Map

```
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: Two-Part of a Quadratic Class Group (ex-53f1e4)

```
> 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
```

37.3.2 Norm Equations

For imaginary quadratic fields, (that is, for quadratic fields $\mathbb{Q}(\sqrt{m})$ 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 Finding Points for details.

NormEquation(F, m): FldQuad, RngIntElt -> BoolElt, SeqEnum

NormEquation(F, m: parameters): FldQuad, RngIntElt -> BoolElt, SeqEnum

NormEquation(O, m): RngQuad, RngIntElt -> BoolElt, SeqEnum

NormEquation(O, m: parameters): RngQuad, RngIntElt -> BoolElt, SeqEnum

```
Factorization: [<RngIntElt, RngIntElt>]                 Default:
All          : BoolElt                                  Default: true
Solutions    : RngIntElt                                Default: All
Exact        : BoolElt                                  Default: false
```

(continues on next page)

(continued from previous page)

```
Ineq         : BoolElt                                    Default: false
NormEquation : Verbose                                    Default: 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 $F = \mathbb{Q}(\sqrt{d})$ 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 The Solution of Modular Equations). 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(s, i): MonStgElt, RngIntElt`).

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 *Solving Norm Equations* for a description.

Example: Norm Equation in a Quadratic Order (ex-6e8332)

```
> d := 302401481761723680;
> m := 76814814791186002463716;
> Q<z> := QuadraticField(-d);
> O<w> := sub< MaximalOrder(Q) | 6 >;
> f, s := NormEquation(O, m);
> s, Norm(s[1]);
406 + 1008*w 76814814791186002463716
```

37.4 Special Element Operations

Several functions are available only for elements of certain maximal orders.

The maximal orders of `QuadraticField(d)` for $d = -1, -2, -3, -7, -11, 2, 3, 5$ and 13 are Euclidean with respect to the `Norm`. The division algorithm, Euclidean algorithm, GCD, and LCM are available for these orders only. Below, we will refer to these orders as the "special Euclidean orders".

37.4.1 Division Algorithm

`a div b: RngQuadElt, RngQuadElt -> RngQuadElt`

This operation is available for all orders, but its behaviour depends on whether the order R containing a and b is one of the special Euclidean orders listed above.

When R is not a special Euclidean order, this returns the exact quotient of a and b. An error results if a is not exactly divisible by b in R.

When R is a special Euclidean order, this performs the division algorithm in R. It finds uniquely determined elements q and r in R such that $a = qb + r$ and $\mathtt{Norm}(r) < \mathtt{Norm}(b)$, and returns q.

`a mod b: RngQuadElt, RngQuadElt -> RngQuadElt`

This operation requires a and b to belong to one of the special Euclidean orders listed above. It performs the same calculation as `div`, and returns the remainder r.

`GreatestCommonDivisor(a, b): RngQuadElt, RngQuadElt -> RngQuadElt`

`Gcd(a, b): RngQuadElt, RngQuadElt -> RngQuadElt`

`GCD(a, b): RngQuadElt, RngQuadElt -> RngQuadElt`

The greatest common divisor of a and b, which must be elements of one of the special Euclidean orders listed above.

`LeastCommonMultiple(a, b): RngQuadElt, RngQuadElt -> RngQuadElt`

`Lcm(a, b): RngQuadElt, RngQuadElt -> RngQuadElt`

`LCM(a, b): RngQuadElt, RngQuadElt -> RngQuadElt`

The least common multiple of a and b, which must be elements of one of the special Euclidean orders listed above.

`Modexp(a, e, n): RngQuadElt, RngInt, RngQuadElt -> RngQuadElt`

This returns $a^e \bmod n$, where a and n must be elements of one of the special Euclidean orders listed above.

37.4.2 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): RngQuadElt -> SeqEnum, Tup`

`Factorisation(n): RngQuadElt -> SeqEnum, Tup`

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): RngQuadElt, RngIntElt -> SeqEnum, SeqEnum, Tup`

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 $(1+\sqrt{-3})/2$).

37.4.3 Conjugates

`ComplexConjugate(a): FldQuadElt -> FldQuadElt`

`ComplexConjugate(a): RngQuadElt -> RngQuadElt`

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 $\mathbb{Q}(\sqrt{d})$.

`Conjugate(a): FldQuadElt -> FldQuadElt`

`Conjugate(a): RngQuadElt -> RngQuadElt`

The conjugate $x - y\sqrt{d}$ of $a = x + y\sqrt{d}$ in the quadratic field $\mathbb{Q}(\sqrt{d})$.

37.4.4 Other Element Functions

For the ring of integers of $\mathbb{Q}(\mathrm{i})$ the biquadratic residue symbol (generalizing the Legendre symbol) is available.

`BiquadraticResidueSymbol(a, b): RngQuadElt, RngQuadElt -> RngQuadElt`

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): RngQuadElt -> RngQuadElt`

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: Primes with 2 as a Biquadratic Residue (ex-c87bf6)

The following example checks for primes p with $65 \leq p \leq 1000$ 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 $\mathbb{Q}(i)$ 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 }
```

37.5 Special Functions for Ideals

Ideals of orders of quadratic fields inherit from ideals of orders of number fields (see Section *Ideals and Quotients* for the list of general functions and operations available). However there is also a correspondence between quadratic ideals and binary quadratic forms (see Chapter Binary Quadratic Forms). The following functions make use of that correspondence.

`Content(I): RngQuadFracIdl -> RngQuadFracIdl`

The content of the ideal.

`Conjugate(I): RngQuadFracIdl -> RngQuadFracIdl`

The conjugate of the ideal.

`Discriminant(I): RngQuadFracIdl -> RngIntElt`

The discriminant of the quadratic form associated with I.

`QuadraticForm(I): RngQuadFracIdl -> QuadBinElt`

The quadratic form associated with I.

`Ideal(f): QuadBinElt -> RngQuadIdl`

The quadratic ideal with associated quadratic form f.

`Reduction(I): RngQuadFracIdl -> RngQuadFracIdl`

The quadratic ideal with associated quadratic form which is a reduction of the quadratic form associated to I.

37.6 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.

Chapter 38

Cyclotomic Fields

38.1 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 *Number Fields*.

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.

38.2 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.

38.2.1 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): RngIntElt -> FldCyc`

`Sparse: Boolean`	`Default: false`

Given a positive integer m, create the field obtained by adjoining the m-th roots of unity to $\mathbb{Q}$. 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): RngIntElt -> RngUPolElt`

Given a positive integer m, create the cyclotomic polynomial of order m. This function is equivalent to `DefiningPolynomial(CyclotomicField(m))`.

`MinimalField(a): FldRatElt -> FldRat`

`MinimalField(a): FldCycElt -> FldCyc`

`MinimalField(a): RngCycElt -> RngCyc`

`MinimalCyclotomicField(a): FldRatElt -> FldRat`

`MinimalCyclotomicField(a): FldCycElt -> FldCyc`

`MinimalCyclotomicField(a): RngCycElt -> RngCyc`

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.

`MinimalField(S): [ FldCycElt ] -> FldCyc`

`MinimalField(S): { FldCycElt } -> FldCyc`

`MinimalField(S): [ RngCycElt ] -> RngCyc`

`MinimalField(S): { RngCycElt } -> RngCyc`

`MinimalCyclotomicField(S): [ FldCycElt ] -> FldCyc`

`MinimalCyclotomicField(S): Setq -> FldCyc`

`MinimalCyclotomicField(S): { FldCycElt } -> FldCyc`

`MinimalCyclotomicField(S): [ RngCycElt ] -> RngCyc`

`MinimalCyclotomicField(S): { RngCycElt } -> RngCyc`

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: Dense and Sparse Cyclotomic Fields (ex-644033)

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
```

38.2.2 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 *Creation of Elements*.

`RootOfUnity(n): RngIntElt -> FldCycElt`

Create the n-th root of unity ζ_n in $\mathbb{Q}(\zeta_n)$.

`RootOfUnity(n, K): RngIntElt, FldCyc -> FldCycElt`

Given a cyclotomic field $K = \mathbb{Q}(\zeta_m)$ 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): FldCycElt`
`Minimize(~a): FldCycElt`
`Minimise(~a): RngCycElt`
`Minimize(~a): RngCycElt`

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 $\mathbb{Q}$ or $\mathbb{Z}$.

`Minimise(~s): [ FldCycElt ]`
`Minimize(~s): [ FldCycElt ]`
`Minimise(~s): [ RngCycElt ]`
`Minimize(~s): [ RngCycElt ]`

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 $\mathbb{Q}$ or $\mathbb{Z}$.

`Minimise(a): FldCycElt -> RngElt`
`Minimize(a): FldCycElt -> RngElt`
`Minimise(a): RngCycElt -> RngElt`
`Minimize(a): RngCycElt -> RngElt`

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 $\mathbb{Q}$ or $\mathbb{Z}$.

`Minimise(s): { FldCycElt } -> { RngElt }`
`Minimize(s): { FldCycElt } -> { RngElt }`
`Minimise(s): { RngCycElt } -> { RngElt }`
`Minimize(s): { RngCycElt } -> { RngElt }`

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 $\mathbb{Q}$ or $\mathbb{Z}$.

38.3 Structure Operations

In cyclotomic fields the generic ring functions are supported (see Chapter Introduction to Rings). 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 *Creation Functions* and Section *Structure Operations*.

38.3.1 Invariants

`Conductor(K): FldCyc -> RngIntElt, [RngIntElt]`

The smallest n such that the field K is contained in $\mathbb{Q}(\zeta_n)$; 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): FldCyc -> RngIntElt`
`CyclotomicOrder(K): FldRat -> RngIntElt`

The value of m for the cyclotomic field $\mathbb{Q}(\zeta_m)$. Note that this will be the m with which the cyclotomic field was created.

`CyclotomicAutomorphismGroup(K): FldCyc -> GrpAb, Map`

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): FldCyc, FldCyc -> FldNum`

Given two cyclotomic fields $k \subseteq K$ a number field L/k is computed that is isomorphic to K.

38.4 Element Operations

For the full range of operations for elements of a number field or order see Section *Element Operations*.

38.4.1 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): FldCycElt -> BoolElt

IsReal(a): RngCycElt -> BoolElt

Whether the cyclotomic field or ring element a is a real number, i.e., if it is invariant under the complex conjugation.

38.4.2 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): FldCycElt -> FldCycElt

ComplexConjugate(a): RngCycElt -> RngCycElt

The complex conjugate of cyclotomic field or ring element a.

Conjugate(a, n): FldCycElt, RngIntElt -> FldCycElt

Conjugate(a, n): RngCycElt, RngIntElt -> RngCycElt

The image under the map $\zeta \mapsto \zeta^n$. The second argument (n) must be coprime to the conductor.

Conjugate(a, r): FldCycElt, FldCycElt -> FldCycElt

Conjugate(a, r): RngCycElt, RngCycElt -> RngCycElt

The conjugate of the element $a \in \mathbb{Q}(\zeta_m)$ or its order, obtained by applying the field automorphism $\zeta_m \mapsto r$ where r is a primitive root of unity.

Example: Gaussian Periods (ex-591c1c)

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 $\mathbb{Q}(\zeta_l)$. 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,
}
```

Chapter 39

Number Fields and Orders

39.1 Introduction

This chapter deals with (general) number fields, and orders and ideals in number fields.

Special types of number fields (quadratic and cyclotomic) have special features described in the corresponding chapters (*Quadratic Fields* and *Cyclotomic Fields*). All the general features here apply to those types too.

39.1.1 Types

Number fields have type `FldNum`, with subtypes `FldQuad` and `FldCyc`. Their orders have types `RngOrd`, `RngQuad`, `RngCyc` respectively. The `FieldOfFractions` of an order is not the number field, rather it is an isomorphic field that has the same basis as the order (for speed); it has type `FldOrd`. The container type `FldAlg` is the union of `FldNum` and `FldOrd` (this is used in signatures that allow both).

Elements in fields and orders have corresponding types: `FldNumElt`, `FldAlgElt`, `RngQuadElt`, and so on.

Extended types are used to indicate the coefficient ring (i.e. base ring). An absolute extension is a `FldNum[FldRat]`, a relative extension is a `FldNum[FldNum]`, similarly for `RngOrd[RngInt]` and so on.

The `RationalField()` is *not* a number field (`FldRat` is not a subtype of `FldNum`). However, one can create `RationalsAsNumberField()`.

A number field (`FldNum`) is an algebraic extension of finite degree over a base field, which may be a number field k or $\mathbb{Q}$. It is constructed as $K = k[t]/(f(t))$ where f is an irreducible polynomial in $k[t]$. A field may also be constructed as a multivariate quotient $K = k[s_1, \ldots, s_n]/(f_1(s_1), \ldots, f_n(s_n))$ where each polynomial is univariate (although the internal representation of this is as a tower of extensions).

Orders can be constructed in two main ways:

- as finite extensions E of a (maximal) order m (or of $\mathbb{Z}$) by adjoining a root of a monic integral polynomial $f \in m[t]$
- as a transformation (i.e. change of basis) of another 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 `Pseudo Matrices` similar to the way modules over Dedekind rings (`ModDed`, `Pseudo Matrices`) 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 field of fractions exists to serve as the parent of fields elements represented in that basis, and support fractional ideals.

An proper ideal of an order has type `RngOrdIdl`, a fractional ideal has type `RngOrdIdlFrac`. An ideal does not play the role of parent to elements in it (their parent is the order or the field of fractions); in

particular, there is no coercion of elements into ideals. Sometimes one needs the parent structure of all ideals or all fractional ideals: these are created by `PowerIdeal` of the `RngOrd` or the `FldOrd`.

The conventions regarding the basis of an order can be confusing. For relative extensions, an order O is not necessarily a free module over its base ring R and therefore a relative integral basis may not exist [Poh]. Therefore, the module structure of O over R is given by `PseudoBasis(Module(O))` which consists of `Basis(O)` together with `CoefficientIdeals(O)`. The elements in `Basis(O)` are not necessarily contained in O; their parent is always `FieldOfFractions(O)`, which has the same basis. Note that `O.i` is `Basis(O)[i]`. For an element x of O, `Eltseq(x)` returns a sequence of coefficients giving x in terms of the basis – these coefficients are elements of `FieldOfFractions(R)`.

In this chapter, the phrase *algebraic field* will refer to a `FldAlg`, i.e. either a number field (`FldNum`) or a field of fractions (`FldOrd`).

39.2 Acknowledgement

The number field module in Magma has mostly developed out of the Kant/Kash system (Kant-V4) [KAN97], [KAN00], developed by the group of M. Pohst at TU Berlin.

39.3 Creation Functions

This section describes the main ways to construct number fields and orders, as well as creation of elements and homomorphisms.

39.3.1 Creation of Algebraic Fields

A field may be created as an *absolute extension* (absolute field), i.e. an extension of Q by an irreducible polynomial, or as a *relative extension* (relative field), i.e. an extension of another field K by an irreducible polynomial over K. See also Section *Creation of Structures* for quadratic fields and Section *Creation Functions* for cyclotomic fields (these are the special subtypes of number fields in Magma). Some other constructions (which involve nontrivial computations) are the composite formed by two or more given fields, and the splitting field of a given polynomial or field.

`NumberField(f): RngUPolElt -> FldNum`

```
Check            : BoolElt                    Default: true
DoLinearExtension: BoolElt                    Default: false
Global           : BoolElt                    Default: false
```

Given an irreducible polynomial f of degree $n \geq 1$ over $K = \mathbb{Q}$ 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 *Creation of Number Fields*.

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() -> FldNum`

`QNF() -> FldNum`

This creates a number field isomorphic to $\mathbb{Q}$. It is equivalent to `NumberField(x-1 : DoLinearExtension)`, where x is `PolynomialRing(Rationals()).1`.

The result is a field isomorphic to $\mathbb{Q}$, but regarded by Magma as a number field (while $\mathbb{Q}$ 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): [ RngUPolElt ] -> FldNum`

```
Check            : BoolElt                    Default: true
DoLinearExtension: BoolElt                    Default: false
Abs              : BoolElt                    Default: false
```

Let K be a possibly trivial algebraic extension of $\mathbb{Q}$. 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 *Creation of Number Fields*.

`ext< F | s1, ..., sn >: FldAlg, RngUPolElt, ..., RngUPolElt -> FldAlg`

`ext<F | s>: FldAlg, [RngUPolElt] -> FldAlg`

`ext< Q | s1, ..., sn >: FldRat, RngUPolElt, ..., RngUPolElt -> FldNum`

`ext< Q | s >: FldRat, [RngUPolElt] -> FldNum`

```
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 $\mathbb{Q}$ 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): Rng, RngIntElt, RngElt -> FldAlg`

```
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): FldAlg -> FldAlg, SeqEnum`

`NormalClosure(F): FldAlg -> FldAlg, SeqEnum`

```
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): RngUPolElt -> FldAlg`

Given an irreducible polynomial f over $\mathbb{Z}$, return its splitting field.

`SplittingField(L): [RngUPolElt] -> FldNum, [FldNumElt]`

```
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 $\mathbb{Q}$. If in addition `Opt` is also `true`, then an optimized representation of K is computed as well.

`sub< F | e1, ..., en >: FldAlg, FldAlgElt, ..., FldAlgElt -> FldAlg, Map`

`sub< F | S >: FldAlg, SeqEnum -> FldAlg, Map`

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): FldAlg, FldAlg -> SeqEnum`

`CompositeFields(F, L): FldAlg, FldAlg -> SeqEnum`

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): FldAlg, FldAlg -> FldAlg`

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): FldAlg, FldAb -> FldAlg`

For a 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): FldAlg -> FldAlg, Map`

`OptimisedRepresentation(F): FldAlg -> FldAlg, Map`

```
Al: MonStgElt                    Default: "Fast"
```

`OptimizedRepresentation(F, d): FldAlg, RngIntElt -> FldAlg, Map`

`OptimisedRepresentation(F, d): FldAlg, RngIntElt -> FldAlg, Map`

Given an algebraic field F with ground field $\mathbb{Q}$, 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 as well as a map from F into L; otherwise F will be returned. If `Al` is set to `Fast`, a fast randomised approach is used. Further, it can be set to `Polredabs` to determine a unique defining polynomial with the same selection criteria as implemented in pari. Setting `Al` to `Old` calls the old algorithm described below.

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: Optimized Representation of a Field (ex-509ba1)

Some results of `OptimizedRepresentation` are shown. Note that `OptimizedRepresentation` is a random algorithm, and may return different results for the same input.

```
> _<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);
true
> f := DefiningPolynomial(K);
// f is an element of Q[x], so Discriminant(f) is in Q not Z
> Z := IntegerRing();
> Factorization(Z !  Discriminant(f));
[ <2, 18>, <5, 8>, <41, 2> ]
> g := DefiningPolynomial(L);
> 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.

39.3.2 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 $\mathcal{E}_K = \mathbb{Z}[\alpha]$ of $K = \mathbb{Q}(\alpha) \cong \mathbb{Q}[X]/f(X)$, 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): RngUPolElt -> RngOrd

```
Check: BoolElt                    Default: true
```

Given an irreducible non-constant monic integral polynomial $f \in R[X]$, return the equation order $E = \mathbb{R}[X]/f(X)$ corresponding to f. If the optional parameter **Check** is set to **false** then the polynomial will not be checked for irreducibility.

EquationOrder(S): [RngUPolElt] -> RngOrd

```
Abs              : BoolElt                    Default: false
DoLinearExtension: BoolElt                    Default: false
Check            : BoolElt                    Default: true
Global           : BoolElt                    Default: false
```

The equation order of *NumberField*(S).

EquationOrder(K): FldNum -> RngOrd

Return the equation order corresponding to the polynomial with which the number field K was defined. The field 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): RngOrd -> RngOrd

Provided the order O is not an equation order, O is a transformation of some order O'. This function returns O'.

EquationOrder(O): RngOrd -> RngOrd

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))). The order O must have a monic defining polynomial for the equation order to exist.

Integers(O): RngOrd -> RngOrd

RingOfIntegers(O): RngOrd -> RngOrd

IntegerRing(O): RngOrd -> RngOrd

Returns the ring of integers in the order O, ie. O itself.

Example: Equation Order and Ring of Integers (ex-4c8125)

Once a number field $K = \mathbb{Q}(\alpha)$ has been created, one can obtain the equation order $E = \mathbb{Z}[\alpha]$ 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 *Orders and Ideals* where a maximal order is created).

sub< O | a_1, ..., a_r **>**: RngOrd, RngOrdElt, ..., RngOrdElt -> RngOrd

Create the suborder of the order O generated (as an algebra over $\mathbb{Z}$) by the elements $a_1, \ldots, a_r \in O$, that is, create $\mathbb{Z}[a_1, \ldots, a_r]$. 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 **>**: RngOrd, RngOrdElt, ..., RngOrdElt -> RngOrd

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< Z | f **>**: RngInt, RngUPolElt -> RngOrd

ext< O | f **>**: RngOrd, RngUPolElt -> RngOrd

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): RngOrd -> FldOrd

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): FldOrd -> RngOrd`

The order of which F was created as its field of fractions. This function is an inverse to `FieldOfFractions`.

`NumberField(O): RngOrd -> FldNum`

The number field of an order is recursively defined by:

1. the number field of **Z** is **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): FldOrd -> FldNum`

The number field of `Order(F)` for a field of fractions F.

Example: Bases of an Order and Its Fraction Field (ex-94a3f9)

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,
```

(continues on next page)

(continued from previous page)

```
    $.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): RngOrd -> BoolElt, RngOrd, Map

OptimisedRepresentation(O): RngOrd -> BoolElt, RngOrd, Map

OptimizedRepresentation(O, d): RngOrd, RngIntElt -> BoolElt, RngOrd, Map

OptimisedRepresentation(O, d): RngOrd, RngIntElt -> BoolElt, RngOrd, Map

Given an order O with ground ring $\mathbb{Z}$, this function will attempt to find an isomorphic order OO with discriminant the same as O with a better defining polynomial than the one used to define O. If such a polynomial is found then OO is returned; otherwise only **false** is 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) $[OO : E_{OO}]$ of the equation order E_{OO} of OO in OO.

O + P: RngOrd, RngOrd -> RngOrd

Add two orders O and P having the same equation order. Computes the smallest common over order.

O **meet** P: RngOrd, RngOrd -> RngOrd

The intersection of two orders O and P having the same equation order.

AsExtensionOf(O, P): RngOrd, RngOrd -> RngOrd

Return the order O as a transformation of the order P where O and P have the same coefficient ring.

Order(O, T, d): RngOrd, AlgMatElt, RngIntElt -> RngOrd

```
Check: BoolElt                    Default: true
```

Let O be an absolute order with basis $b_1, \ldots, n_n$, $T = (T_{i,j}) \in \mathrm{GL}(n, \mathbb{Q}) \cap \mathrm{Mat}(n, \mathbb{Z})$ and $d \in \mathbb{N}$. This function creates the order with basis $(1/d \sum_{j=1}^{n} T_{i,j} b_j)_{i \leq i \leq n}$. The parameter **Check** can be set to **false** when the order is large to avoid checking that the result actually is an order. Note that this can result in a non-order being constructed which may cause errors later.

Order(O, M): RngOrd, ModDed -> RngOrd

```
NFBasis: BoolElt                  Default: true
```

Order(O, M): RngOrd, PMat -> RngOrd

```
Check: BoolElt                    Default: true
```

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 the parameter **Check** is set to **false**, then it will not be checked that the result actually is an order (potentially expensive). Note that this can result in a non-order being constructed which may cause errors later. If the parameter **NFBasis** is set to **false** then the **PseudoGenerators** of

the module M will be used rather than the `PseudoBasis`, however these pseudo generators must also be a pseudo basis.

`Order([ e1, ... en ]): [FldAlgElt] -> RngOrd`

`Verify : BoolElt`	`Default: true`
`IsBasis: BoolElt`	`Default: false`

Given n elements $e_1, \ldots, e_n$ in an algebraic extension field F over $\mathbb{Q}$ 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 `IsBasis` 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. Note that setting `IsBasis` to `true` can result in a non-order being constructed if the elements in the sequence are not a basis which may cause errors later. By default, products of the generators will be added until the module is closed under multiplication.

If `IsBasis` is set to `true` to specify the basis of the resulting order rather than avoid the expense of the multiplicative closure computation, it can be checked that the result O is an order using `Order(SubOrder(O), Matrix(CoefficientRing(O), M*d), d) where d is Denominator(M) where M is BasisMatrix(O);`. If O is not an order this will cause an error.

39.3.3 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 ([Bai96, Coh93, 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].

It is possible to compute the maximal order of an absolute extension of $\mathbb{Z}$ or $\mathbb{Q}$ defined by one monic, integral polynomial using the Montes Algorithm [Sta18]. The Montes algorithm can be selected using `Al := "Montes"` when the input is a field or an order. For more on the Montes Algorithm see Section *The Montes Algorithm*.

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 in this case `Al` cannot be specified. This algorithm is based on [BL94c], Theorems 1.2 and 7.6.

Since version 2.30, a new driver combining the Montes and Buchmann-Lenstra methods is available for computing the maximal order of an absolute number field of degree greater than two. The parameter `Hint` can be used to supply a sequence of integers that may contain factors of the discriminant. Setting `TimeLimit` to a positive value limits integer factorization using runtime estimates calibrated for an i7

processor. This is an approximate limit: actual runtimes can vary substantially, even on the reference hardware. A second return value indicates whether the computation was interrupted by the time limit: it is set to true if the computation ran to completion. Finally, the [Mul25] factorization can be used by setting `Mulder` to `true`.

`MaximalOrder(O): RngOrd -> RngOrd, BoolElt`

`MaximalOrder(F): FldAlg -> RngOrd, BoolElt`

```
Al          : MonStgElt                 Default: "Auto"
Discriminant: Any                       Default:
Ramification: SeqEnum                   Default:
Hint        : SeqEnum                   Default:
TimeLimit   : RngElt                    Default: 0
Mulder      : BoolElt                   Default: false
MaximalOrder: Verbose                   Default: 5
```

`IntegerRing(F): FldAlg -> RngOrd`

`Integers(F): FldAlg -> RngOrd`

`RingOfIntegers(F): FldAlg -> RngOrd`

```
Al          : MonStgElt                 Default: "Auto"
Discriminant: Any                       Default:
Ramification: SeqEnum                   Default:
MaximalOrder: Verbose                   Default: 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, *Maximal Orders*. The variants that support `TimeLimit` return, as a second value, `true` if the computation ran to completion. This is only available for absolute number fields.

`MaximalOrder(f): RngUPolElt -> RngOrd, BoolElt`

```
Check       : BoolElt                   Default: true
Al          : MonStgElt                 Default: "Auto"
Discriminant: Any                       Default:
Ramification: SeqEnum                   Default:
Hint        : SeqEnum                   Default:
TimeLimit   : RngElt                    Default: 0
Mulder      : BoolElt                   Default: false
MaximalOrder: Verbose                   Default: 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: Maximal Order with Known Ramification `(ex-474b81)`

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];
```

(continues on next page)

(continued from previous page)

```
> 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
true
> 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
true
> 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 -
> 3987191930969244752361 6*t^3 + 60126791399546070679893112*t^2 +
> 778441185338527286987510 40*t - 831734981995068547514587013 76;
> 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 - 3987191930969244752361 6*x^3 +
    60126791399546070679893112*x^2 + 778441185338527286987510 40*x -
    831734981995068547514587013 76 over Z
Time: 0.730
true
> 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 - 3987191930969244752361 6*x^3 +
    60126791399546070679893112*x^2 + 778441185338527286987510 40*x -
    831734981995068547514587013 76 over Z
Time: 0.840
true
> f13 := t^15 - 114*t^14 + 282185319*t^13 + 1247857228852*t^12 -
> 35114805704965233*t^11 - 14152433779643338782 6*t^10 +
> 260458498044226402874400 9*t^9 + 14153948932132918272984150384*t^8 -
> 17827307724835336994132762847955 2*t^7 - 11429535068213909144192605644943047 68
> *t^6 + 15975069142211276963134599495014990639616*t^5 +
> 33516684438303088018217308253251277376159744*t^4 -
> 61758977710820371623239637245361930955447125606 4*t^3 -
> 39756141244506691954546176235488463150180617486 3360*t^2 +
> 22666571829085475706482454642151923578020471016281866 24*t
> - 13022224565327602564069162232593069605616574287778141962 24;
> 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 -
    14152433779643338782 6*x^10 + 260458498044226402874400 9*x^9 +
    14153948932132918272984150384*x^8 - 17827307724835336994132762847955 2*x^7 -
    11429535068213909144192605644943047 68*x^6 +
```

(continues on next page)

(continued from previous page)

```
    1597506914221127696313459949501499063961 6*x^5 +
    3351668443830308801821730825325127737615 9744*x^4 -
    6175897771082037162323963724536193095544 71256064*x^3 -
    3975614124450669195454617623548846315018 06174863360*x^2 +
    2266657182908547570648245464215192357802 04710162818 6624*x -
    1302222456532760256406916223259306960561 657428777814196224 over Z
Time: 0.270
true
```

Example: Maximal Order with new driver (ex-474b82)

The usage of the new driver is shown in the next examples.

```
> R<t> := PolynomialRing(Integers());
> f13 := t^15 - 114*t^14 + 282185319*t^13 + 1247857228852*t^12 -
> 3511480570496523 3*t^11 - 1415243377964333878 26*t^10 +
> 2604584980442264028744009*t^9 + 1415394893213291827298415038 4*t^8 -
> 1782730772483533699413276284795 52*t^7 - 1142953506821390914419260564494304768
> *t^6 + 1597506914221127696313459949501499063961 6*t^5 +
> 3351668443830308801821730825325127737615 9744*t^4 -
> 6175897771082037162323963724536193095544 71256064*t^3 -
> 3975614124450669195454617623548846315018 06174863360*t^2 +
> 2266657182908547570648245464215192357802 04710162818 6624*t
> - 1302222456532760256406916223259306960561 657428777814196224;
> time MaximalOrder(f13);
Maximal Order of Equation Order with defining polynomial x^15 - 114*x^14 +␣
↪282185319*x^13 + 1247857228852*x^12 - 3511480570496523 3*x^11
 - 1415243377964333878 26*x^10 + 2604584980442264028744009*x^9 +
 1415394893213291827298415038 4*x^8 - 1782730772483533699413276284795 52*x^7 -
 1142953506821390914419260564494304768*x^6 +
 1597506914221127696313459949501499063961 6*x^5 +
 3351668443830308801821730825325127737615 9744*x^4 -
 6175897771082037162323963724536193095544 71256064*x^3 -
 3975614124450669195454617623548846315018 06174863360*x^2 +
 2266657182908547570648245464215192357802 04710162818 6624*x -
 1302222456532760256406916223259306960561 657428777814196224 over its ground order
true
Time: 0.060
> p1 := 10^50+151; p2 := 10^55+21;
> time MaximalOrder(t^3 - p1*p2 : TimeLimit := 1);
Equation Order with defining polynomial x^3 -
 1000000000000000000000000000000000000000000000151000210000000000000000000\
 0000000000000000000000000003171 over its ground order
false
Time: 0.850
> Discriminant($1);
-270000000000000000000000000000000000000000000000815401134000000000000000000000\
0000000000000000615630424681190700000000000000000000000000000000000025856369\
9591400000000000000000000000000000000000000271491507
> time MaximalOrder(t^3 - p1*p2 : Hint := [p1], TimeLimit := 1);
Maximal Equation Order with defining polynomial x^3 -
 1000000000000000000000000000000000000000000000151000210000000000000000000\
 0000000000000000000000000003171 over its ground order
true
Time: 0.060
> Discriminant($1);
```

(continues on next page)

(continued from previous page)

```
-270000000000000000000000000000000000000000000000815401134000000000000000000000\
000000000000000006156304246811907000000000000000000000000000000000000025856369\
95914000000000000000000000000000000000000000271491507
```

Thus, in some cases the maximal order can be determined without a hard integer factorization. Using `hint` to give partial information about ramified primes can be helpful. Further, in many cases the maximal order is found (but not proven) without a hard factorization.

Orders and Ideals

Orders may be created using ideals of another order. Ideals are discussed in Section *Ideals and Quotients*. The following intrinsics form part of the computation of maximal orders as discussed above in Section *Maximal Orders*.

`pMaximalOrder(O, p): RngOrd, RngIntElt -> RngOrd`

`pMaximalOrder(O, p): RngOrd, RngOrdIdl -> RngOrd`

```
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): RngOrd, RngIntElt -> RngOrdIdl`

`pRadical(O, p): RngOrd, RngOrdIdl -> RngOrdIdl`

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.

$$\{x \in F \mid \mathrm{Tr}(xO) \subseteq p\mathbb{Z}\}.$$

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 [BL94c, Fri00] or at least "good" approximations.

`MultiplicatorRing(I): RngOrdFracIdl -> Rng`

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: Implementing the Round 2 Algorithm (ex-638433)

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
```

39.3.4 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 $\mathbb{Q}$, 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 $\mathbb{Z}$-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: FldAlg, RngElt -> FldAlgElt`

`elt< F | a >: FldAlg, RngElt -> FldAlgElt`

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 ! [a0, a1, ..., am-1]: FldAlg, [RngElt] -> FldAlgElt`

`elt< F | [ a0, a1, ..., am-1 ] >: FldAlg, SeqEnum[RngElt] -> FldAlgElt`

`elt< F | a0, a1, ..., am-1>: FldAlg, RngElt, ..., RngElt -> FldAlgElt`

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: RngOrd, RngElt -> RngOrdElt`

`elt< O | a >: RngOrd, RngElt -> RngOrdElt`

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 ! [a0, a1, ..., am-1]: RngOrd, [ RngElt ] -> RngOrdElt`

`elt< O | [ a0, a1, ..., am-1 ] >: RngOrd, SeqEnum[RngElt] -> RngOrdElt`

`elt< O | a0, a1, ..., am-1 >: RngOrd, RngElt, ..., RngElt -> RngOrdElt`

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): FldAlg, RngIntElt -> FldAlgElt`

`Random(O, m): RngOrd, RngIntElt -> RngOrdElt`

A random element of the algebraic field F or order O. The maximal size of the coefficients is determined by m.

`Random(I, m): RngOrdFracIdl, RngIntElt -> FldOrdElt`

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: Three Ways to Create an Element (ex-759e86)

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);
> // Coordinates of e with respect to Basis(O):
> v := Eltseq(e);
> v;
[ 0, 1, 1, 0 ]
> // The same coordinates build the element directly,
> // both in O and in its field of fractions:
> f := elt< O | v >;
> f eq e;
true
> F<a, b, c, d> := FieldOfFractions(O);
> g := F ! v;
> g eq e;
true
> g;
b + c
```

These constructions would have failed if the element was not in O.

`One(K): FldAlg -> FldAlgElt`

`One(O): RngOrd -> RngOrdElt`

`Identity(K): FldAlg -> FldAlgElt`

`Identity(O): RngOrd -> RngOrdElt`

`Zero(K): FldAlg -> FldAlgElt`

`Zero(O): RngOrd -> RngOrdElt`

`Representative(K): FldAlg -> FldAlgElt`

`Representative(O): RngOrd -> RngOrdElt`

39.3.5 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 >: FldAlg, Rng, RngElt -> Map`

`hom< F -> R | h, r >: FldAlg, Rng, Map, RngElt -> Map`

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 $G = \mathbb{Q}$) and sometimes necessary (if $G \neq \mathbb{Q}$) 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 >: RngOrd, Rng, RngElt -> Map`

`hom< O -> R | h, r >: RngOrd, Rng, Map, RngElt -> Map`

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: Embedding a Quadratic Field in a Quartic Field (ex-d59686)

We show a way to embed the field $\mathbb{Q}(\sqrt{2})$ in $\mathbb{Q}(\sqrt{2}+\sqrt{3})$. 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 >: RngFunOrd, Rng, RngElt, ..., RngElt -> Map

hom< O -> R | m, b_1, ..., b_n >: RngFunOrd, Rng, Map, RngElt, ..., RngElt -> Map

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): Map -> BoolElt

Return whether the vector space homomorphism m is a homomorphism of rings.

39.4 Printing

SetVerbose(s, n): MonStgElt, RngIntElt

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:
```

(continues on next page)

(continued from previous page)

```
prime: 2, prime_bound: 3, algo_flag: 89
----------------------------
use dedekind-test selected
Order is already 2-maximal.
1.818446459232066823483698963560708993786253942768999999999
```

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): BoolElt -> BoolElt`

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`.

39.5 Real Precision

Many number field functions involve calculations over real/complex fields. Sometimes it may be necessary to increase the precision used internally. On the other hand, since the default precision is quite high, some calculations may be sped up by decreasing it – at the user's own risk!

Note however that many 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.

`SetKantPrecision(F, n): FldAlg, RngIntElt`

`SetKantPrecision(F, n, m): FldAlg, RngIntElt, RngIntElt`

`SetKantPrecision(O, n): RngOrd, RngIntElt`

`SetKantPrecision(O, n, m): RngOrd, RngIntElt, RngIntElt`

Set the precision to be used internally for real/complex calculations involving the field F or the order O to be n.

The second integer m sets the precision for some unit calculations (this may become obsolete).

39.6 Structure Operations

In the lists below F usually refers to an algebraic field, K to a number field, and O to an order.

39.6.1 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): FldAlg -> Cat`

`Parent(F): FldAlg -> Pow`

`Category(O): RngOrd -> Cat`

`Parent(O): RngOrd -> Pow`

AssignNames(~K, s): FldNum, [MonStgElt]

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): FldNum, RngIntElt -> FldNumElt

K . i: FldNum, RngIntElt -> FldNumElt

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): FldOrd, [MonStgElt]

Assign the strings in the sequence s to the names of the basis elements of the field of fractions F.

F . i: FldOrd, RngIntElt -> FldOrdElt

Name(F, i): FldOrd, RngIntElt -> FldOrdElt

Return the ith basis element of the field of fractions F.

O . i: RngOrd, RngIntElt -> FldOrdElt

Return the ith basis element of the order O.

39.6.2 Related Structures

Each order and field has other orders and fields which are related to it in various ways.

GroundField(F): FldAlg -> Fld

BaseField(F): FldAlg -> Fld

CoefficientField(F): FldAlg -> Fld

CoefficientRing(F): FldAlg -> Fld

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 $\mathbb{Q}$.

BaseRing(O): RngOrd -> Rng

CoefficientRing(O): RngOrd -> Rng

Given an order O, this returns the order over which O was defined. For an absolute order O this will be the integers $\mathbb{Z}$.

AbsoluteField(F): FldAlg -> FldAlg

Given an algebraic field F, this returns an isomorphic number field L defined as an absolute extension (i.e. over $\mathbb{Q}$).

AbsoluteOrder(O): RngOrd -> RngOrd

Given an order O, this returns an isomorphic order O' defined as an order in an absolute extension (over $\mathbb{Q}$).

SimpleExtension(F): FldAlg -> FldAlg

SimpleExtension(O): RngOrd -> RngOrd

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. (For algorithm, see [Tra76])

RelativeField(F, L): FldAlg, FldAlg -> FldAlg

`RelativeField(Q, L): FldRat, FldAlg -> FldAlg`

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.

`Components(F): FldAlg -> [FldAlg]`

`Components(O): RngOrd -> [RngOrd]`

Given an equation order O or a field of fractions F of an equation order O return the sequence of orders or fields each defined by a defining polynomial of O or F.

Example: Compositum of Two Number Fields (ex-3aa334)

It is often desirable to build up a number field by adjoining several algebraic numbers to $\mathbb{Q}$. The following function returns a number field that is the composite field of two given number fields K and L, provided that $K \cap L = \mathbb{Q}$; if K and L have a common subfield larger than $\mathbb{Q}$ 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 $\mathbb{Q}(\sqrt{2}, \sqrt{3}, \sqrt{5})$, 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
```

(continues on next page)

(continued from previous page)

```
> 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): RngOrd -> RngOrd

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): RngOrd -> RngOrd, AlgMatElt

```
Delta: RngElt                    Default: 0.99
```

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: LLL Reduction of an Order Basis (ex-06710d)

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 (*Implementing the Round 2 Algorithm*) we have

```
> R<x> := PolynomialRing(Integers());
> K := NumberField(x^4-420*x^2+40000);
> E := EquationOrder(K);
```

(continues on next page)

(continued from previous page)

```
> 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]
```

(continues on next page)

(continued from previous page)

```
    |
   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): FldAlg -> RngRat

PrimeField(F): FldAlg -> RngRat

PrimeRing(O): RngOrd -> RngInt

Centre(F): FldAlg -> FldAlg

Centre(O): RngOrd -> RngOrd

Embed(F, L, a): FldAlg, FldAlg, FldAlgElt

```
Overwrite: BoolElt                    Default: false
```

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.

If the addition of this embedding causes an inconsistency with currently known embeddings then the embedding will not be added unless `Overwrite` is set to `true`.

Embed(F, L, a): FldAlg, FldAlg, [FldAlgElt]

```
Overwrite: BoolElt                    Default: false
```

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.

If the addition of this embedding causes an inconsistency with currently known embeddings then the embedding will not be added unless `Overwrite` is set to `true`.

EmbeddingMap(F, L): FldAlg, FldAlg -> Map

Returns the embedding map of F in L if an embedding is known.

HasEmbedding(F, L): FldAlg, FldAlg -> Bool

HasEmbedding(O1, O2): RngOrd, RngOrd -> Bool

```
Print: RngIntElt                      Default: 0
```

Return whether there is an embedding of F into L or $O1$ into $O2$ already installed. For information about the installed embedding set the `Print` parameter > 0 with higher values providing more information than lower.

CanComputeEmbedding(F, L): FldAlg, FldAlg -> Bool

`CanComputeEmbedding(O1, O2): RngOrd, RngOrd -> Bool`

```
Print: RngIntElt                    Default: 0
```

Return whether an embedding of F into L or $O1$ into $O2$ can be computed. For information about the installed embedding set the `Print` parameter > 0 with higher values providing more information than lower.

Example: Embedding One Number Field in Another (ex-a44554)

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
> HasEmbedding(k, l);
false
> CanComputeEmbedding(k, l);
false
> Embed(k, l, l.1);
> l!k.1;
l.1
> HasEmbedding(k, l : Print := 1);
From Equation Order with defining polynomial x^2 - 2 over Z
 to Equation Order with defining polynomial x^2 - 2 over Z
 costs 11 type Composition
true
> HasEmbedding(l, k : Print := 1);
From Equation Order with defining polynomial x^2 - 2 over Z
 to Equation Order with defining polynomial x^2 - 2 over Z
 costs 100 type Inverse of an Inclusion
true
> Embed(l, k, k.1);
> k!l.1;
k.1
> HasEmbedding(l, k : Print := 1);
From Equation Order with defining polynomial x^2 - 2 over Z
 to Equation Order with defining polynomial x^2 - 2 over Z
 costs 11 type Composition
true
```

`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] : Overwrite := true);
> L!l.1;
1/36*(-b^4 - 18*b)
```

Completion(K, P): FldAlg, RngOrdIdl -> FldLoc, Map

Completion(O, P): RngOrd, RngOrdIdl -> RngLoc, Map

comp<K|P**>**: FldAlg, RngOrdIdl -> FldLoc, Map

comp<O|P**>**: RngOrd, RngOrdIdl -> RngLoc, Map

```
Precision: RngIntElt                    Default: 50
```

For an absolute extension K of $\mathbb{Q}$ or O of $\mathbb{Z}$, 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 default 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): FldAlg, PlcNumElt -> FldLoc, Map

```
Precision: RngIntElt                    Default: 50
```

For an absolute extension K over $\mathbb{Q}$ and a (finite) place P, compute the completion at P. The precision and the map are as described for *Completion*.

LocalRing(P, prec): RngOrdIdl, RngIntElt -> RngLoc, Map

The completion of **Order**(P) at the prime ideal P up to default precision **prec**.

Localization(O, P): RngOrd, RngOrdIdl -> RngVal, Map

Given an order O of an algebraic number field and a prime ideal p of O return the localization of O at p and the map from O into the localization.

39.6.3 Representing Fields as Vector Spaces

These functions are used to recreate a number field as an associative algebra or as a vector space. In both cases, the base field may be any subfield.

Algebra(K, J): FldAlg, Fld -> AlgAss, Map

Algebra(K, J, S): FldAlg, Fld, [FldAlgElt] -> AlgAss, Map

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): FldAlg, Fld -> ModTupFld, Map

KSpace(K, J): FldAlg, Fld -> ModTupFld, Map

VectorSpace(K, J, S): FldAlg, Fld, [FldAlgElt] -> ModTupFld, Map

`KSpace(K, J, S): FldAlg, Fld, [FldAlgElt] -> ModTupFld, Map`

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: Minimal Polynomial over a Subfield (ex-4636b1)

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
```

(continues on next page)

(continued from previous page)

```
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
```

39.6.4 Invariants

Some information describing an order can be retrieved.

Characteristic(F): FldAlg -> RngIntElt

Characteristic(O): RngOrd -> RngIntElt

Degree(O): RngOrd -> RngIntElt

Degree(F): FldAlg -> RngIntElt

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): RngOrd -> RngIntElt

AbsoluteDegree(F): FldAlg -> RngIntElt

Given an order O or an algebraic field F, return the absolute degree of O over $\mathbb{Z}$ or F over $\mathbb{Q}$.

Discriminant(O): RngOrd -> RngIntElt

Discriminant(O): RngInt -> RngIntElt

Discriminant(F): FldAlg -> RngIntElt

Discriminant(O): RngOrd -> RngOrdIdl

Discriminant(F): FldAlg -> RngOrdIdl

Given an extension F of $\mathbb{Q}$, 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 $\mathbb{Z}$ 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): RngOrd -> RngIntElt

Given an order O, return the absolute value of the discriminant of O regarded as an order over $\mathbb{Z}$.

AbsoluteDiscriminant(K): FldAlg -> FldRatElt

Given an algebraic field K, return the absolute value of the discriminant of K regarded as an extension of $\mathbb{Q}$.

ReducedDiscriminant(O): RngOrd -> RngIntElt

ReducedDiscriminant(F): FldAlg -> RngIntElt

ReducedDiscriminant(O): RngOrd -> RngOrdIdl

`ReducedDiscriminant(F): FldAlg -> RngOrdIdl`

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): RngOrd -> FldReElt`

`Current: BoolElt`	`Default: false`

`Regulator(K): FldNum -> FldReElt`

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.

The precision of the answer may be controlled by using `SetKantPrecision` for the relevant order.

`RegulatorLowerBound(O): RngOrd -> FldReElt`

`RegulatorLowerBound(K): FldNum -> FldReElt`

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): RngOrd -> RngIntElt, RngIntElt`

`Signature(F): FldAlg -> RngIntElt, RngIntElt`

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): RngOrd -> RngIntElt`

`UnitRank(K): FldNum -> RngIntElt`

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): RngOrd, RngOrd -> RngIntElt`

`Index(O, S): RngOrd, RngOrd -> RngOrdIdl`

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): FldAlg -> RngUPolElt`

`DefiningPolynomial(O): RngOrd -> RngUPolElt`

`DefiningPolynomial(F): FldAlg -> [RngUPolElt]`

`DefiningPolynomial(O): RngOrd -> [RngUPolElt]`

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): RngOrd, RngIntElt -> [ FldReElt ]`

`Zeros(O, n): RngOrd, RngIntElt -> [ FldReElt ]`

`Zeroes(F, n): FldAlg, RngIntElt -> [ FldReElt ]`

`Zeros(F, n): FldAlg, RngIntElt -> [ FldReElt ]`

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: Zeros and Defining Polynomial (ex-afbcf8)

The information provided by `Zeros` and `DefiningPolynomial` is illustrated below.

```
> L := NumberField(x^6+108);
> DefiningPolynomial(L);
x^6 + 108
> Zeros(L, 30);
[ 1.889881574842309747150815910899999999994 +
1.091123635971721403560072614199999999977*i,
1.889881574842309747150815910899999999994 -
1.091123635971721403560072614199999999977*i,  0.E-29 +
2.182247271943442807120145228399999999955*i,  0.E-29 -
2.182247271943442807120145228399999999955*i,
-1.889881574842309747150815910899999999994 +
1.091123635971721403560072614199999999977*i,
-1.889881574842309747150815910899999999994 -
1.091123635971721403560072614199999999977*i ]
> l := NumberField(x^3 - 2);
> DefiningPolynomial(l);
x^3 - 2
> Zeros(l, 30);
[ 1.259921049894873164767210607299999999994,
-0.629960524947436582383605303639109999999 +
1.091123635971721403560072614199999999977*i,
-0.629960524947436582383605303639109999999 -
1.091123635971721403560072614199999999977*i ]
```

`Different(O): RngOrd -> RngOrdIdl`

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): RngOrd -> RngOrdIdl`

The conductor of an order O is the largest ideal of its maximal order that is still contained in O: $\{x \in M \mid xM \subseteq O\}$.

39.6.5 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): RngOrd -> [ FldOrdElt ]`

`Basis(O, R): RngOrd, Rng -> [ RngElt ]`

`Basis(F): FldAlg -> [ FldAlgElt ]`

`Basis(F, R): FldAlg, Rng -> [ RngElt ]`

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.

When O is an extension of another order, the basis is to be interpreted with the corresponding `CoefficientIdeals`.

`IntegralBasis(F): FldAlg -> [ FldAlgElt ]`

`IntegralBasis(F, R): FldAlg, Rng -> [ RngElt ]`

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: Basis over Different Rings (ex-6e96c8)

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): FldAlg -> [FldAlgElt]

Returns an absolute basis for the algebraic field K, i.e. a basis for K as a $\mathbb{Q}$ 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): RngOrd -> AlgMatElt

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): RngOrd, RngOrd -> AlgMatElt, RngIntElt

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): RngOrd -> [RngOrdFracIdl]`

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: Basis and Integral Basis (ex-292651)

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)
]
[
    0.1,
    0.2,
    0.3,
    0.4
]
> 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 $\mathbb{Z}$-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*0.1 - 360/1*0.2 + 800/1*0.4
> K!$1;
y^3 + 7
```

MultiplicationTable(O): RngOrd -> [AlgMatElt]

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): RngOrd -> AlgMatElt

TraceMatrix(F): FldAlg -> AlgMatElt

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: Multiplication Table of an Order (ex-8f1664)

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 ($1 \leq i \leq 4$) 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];
0.2
> 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*0.1 - 7/1*0.2 + 5/1*0.3 + 10/1*0.4
> B[2]*B[3];
-5/1*0.1 - 7/1*0.2 + 5/1*0.3 + 10/1*0.4
```

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]
```

(continues on next page)

(continued from previous page)

```
[ 21 105 173 118]
[  2 215 118 226]
```

39.6.6 Ring Predicates

Orders and algebraic fields can be tested for having several properties that may hold for general rings.

`N eq O: RngOrd, RngOrd -> BoolElt`

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: FldAlg, FldAlg -> BoolElt`

`F eq L: FldRat, FldAlg -> BoolElt`

`F eq L: FldAlg, FldRat -> BoolElt`

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): Rng -> BoolElt`

`IsUnitary(R): Rng -> BoolElt`

`IsFinite(R): Rng -> BoolElt`

`IsOrdered(R): Rng -> BoolElt`

`IsField(R): Rng -> BoolElt`

These predicates are clear.

`IsNumberField(R): . -> BoolElt`

Returns `true` iff `Type(R)` matches `FldNum`.

`IsAlgebraicField(R): Any -> BoolElt`

Returns `true` iff `Type(R)` matches `FldAlg` (i.e. either `FldNum` or `FldOrd`).

`IsEuclideanDomain(F): FldAlg -> BoolElt`

This is not a check for euclidean number fields. This function will always return an error.

`IsSimple(F): FldAlg -> BoolElt`

`IsSimple(O): RngOrd -> BoolElt`

Checks if the field F or the order O is defined as a simple extension over the base ring.

`IsPID(F): FldAlg -> BoolElt`

`IsUFD(F): FldAlg -> BoolElt`

`IsPrincipalIdealRing(F): FldAlg -> BoolElt`

Always `true` for fields.

`IsPID(O): RngOrd -> BoolElt`

`IsUFD(O): RngOrd -> BoolElt`

`IsPrincipalIdealRing(O): RngOrd -> BoolElt`

Always `false` for orders. Even if the class number is 1, orders are not considered to be PIDs.

`IsDomain(R): FldAlg -> BoolElt`

`F ne L: FldAlg, FldAlg -> BoolElt`

`O ne N: RngOrd, RngOrd -> BoolElt`

`O subset P: RngOrd, RngOrd -> BoolElt`

`K subset L: FldAlg, FldAlg -> BoolElt`

`K subset L: FldRat, FldAlg -> BoolElt`

`HasComplexConjugate(K): FldAlg -> BoolElt, Map`

This function returns `true` if there is an automorphism in the field K that acts like complex conjugation.

`ComplexConjugate(x): FldAlgElt -> FldAlgElt`

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.

39.6.7 Order Predicates

Since orders are rings with additional properties, special predicates are applicable.

`IsEquationOrder(O): RngOrd -> BoolElt`

This returns `true` if the basis of the order O is an integral power basis, `false` otherwise.

`IsMaximal(O): RngOrd -> BoolElt`

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): RngOrd -> BoolElt`

Returns `true` iff the order O is a constructed as an absolute extension of $\mathbb{Z}$.

`IsWildlyRamified(O): RngOrd -> BoolElt`

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): RngOrd -> BoolElt`

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): RngOrd -> BoolElt`

Returns `true` iff the order O is unramified at the *finite* places.

39.6.8 Field Predicates

Here all the predicates that are specific to algebraic fields are listed.

`IsIsomorphic(F, L): FldAlg, FldAlg -> BoolElt, Map`

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): FldAlg, FldAlg -> BoolElt, Map`

`IsSubfield(F, L): FldRat, FldAlg -> BoolElt, Map`

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): FldAlg -> BoolElt`

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): FldAlg -> BoolElt`

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): FldAlg -> BoolElt`

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): FldAlg -> BoolElt`

Returns `true` iff the algebraic field K is a constructed as an absolute extension of $\mathbb{Q}$.

`IsWildlyRamified(K): FldAlg -> BoolElt`

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): FldAlg -> BoolElt`

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): FldAlg -> BoolElt`

Returns `true` iff the algebraic field K is unramified at the *finite* places.

`IsQuadratic(K): FldAlg -> BoolElt, FldQuad`

If the number field K is quadratic, return `true` and an isomorphic quadratic field.

`IsTotallyReal(K): FldAlg -> BoolElt`

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.

39.6.9 Setting Properties of Orders

Several properties of orders can be set to be known and/or to have a given value.

`SetOrderMaximal(O, b): RngOrd, BoolElt`

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): RngOrd, RngOrdElt, RngIntElt`

Set the torsion unit of the order O to be the element e with order r.

`SetOrderUnitsAreFundamental(O): RngOrd`

This tells Magma to assume that the currently known unit group for the order O is the full unit group, in all subsequent calculations involving `NumberField(O)`. (This can only be invoked after a unit group has been computed.)

39.7 Element Operations

39.7.1 Parent and Category

`Parent(a): FldAlgElt -> FldAlg`

`Parent(w): RngOrdElt -> RngOrd`

`Category(a): FldAlgElt -> Cat`

`Category(w): RngOrdElt -> Cat`

39.7.2 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 *Ideal Arithmetic*.

`+ a: FldAlgElt -> FldAlgElt`
`+ w: RngOrdElt -> RngOrdElt`
`- a: FldAlgElt -> FldAlgElt`
`- w: RngOrdElt -> RngOrdElt`
`a + b: FldAlgElt, FldAlgElt -> FldAlgElt`
`w + v: RngOrdElt, RngOrdElt -> RngOrdElt`
`a - b: FldAlgElt, FldAlgElt -> FldAlgElt`
`w - v: RngOrdElt, RngOrdElt -> RngOrdElt`
`a * b: FldAlgElt, FldAlgElt -> FldAlgElt`
`w * v: RngOrdElt, RngOrdElt -> RngOrdElt`
`a / b: FldAlgElt, FldAlgElt -> FldAlgElt`
`w / v: RngOrdElt, RngOrdElt -> FldOrdElt`
`a ^ k: FldAlgElt, RngIntElt -> FldAlgElt`
`w ^ k: RngOrdElt, RngIntElt -> RngOrdElt`
`w div v: RngOrdElt, RngOrdElt -> RngOrdElt`

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): RngOrdElt, RngIntElt, RngIntElt -> RngOrdElt`

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): RngOrdElt -> RngOrdElt`
`Sqrt(a): FldAlgElt -> FldAlgElt`
`SquareRoot(a): RngOrdElt -> RngOrdElt`
`SquareRoot(a): FldAlgElt -> FldAlgElt`

Returns the square root of the element a if it exists in the order or field containing a.

`Root(a, n): RngOrdElt, RngIntElt -> RngOrdElt`
`Root(a, n): FldAlgElt, RngIntElt -> FldAlgElt`

Returns the n-th root of the element a if it exists in the order or field containing a.

`IsPower(a, k): FldAlgElt, RngIntElt -> BoolElt, FldAlgElt`
`IsPower(a, k): RngOrdElt, RngIntElt -> BoolElt, RngOrdElt`
`IsSquare(a): FldAlgElt -> BoolElt, FldAlgElt`
`IsSquare(a): RngOrdElt -> BoolElt, RngOrdElt`

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): FldAlgElt -> RngIntElt`
`Denominator(a): RngOrdElt -> RngIntElt`

Returns the denominator of the element a, that is the least common multiple of the denominators of the coefficients of a.

`Numerator(a): FldAlgElt -> RngIntElt`

Returns the numerator of the element a, that is the element multiplied by its denominator.

`Qround(E, M): FldAlgElt, RngIntElt -> FldAlgElt`

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.

39.7.3 Equality and Membership

Elements may also be tested for whether they lie in an ideal of an order. See Section *Predicates on Ideals*.

`a eq b: FldAlgElt, FldAlgElt -> BoolElt`

`w eq v: RngOrdElt, RngOrdElt -> BoolElt`

`a ne b: FldAlgElt, FldAlgElt -> BoolElt`

`w ne v: RngOrdElt, RngOrdElt -> BoolElt`

`a in F: FldAlgElt, FldAlg -> BoolElt`

`w in O: RngOrdElt, RngOrd -> BoolElt`

39.7.4 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): FldAlgElt -> BoolElt`
`IsIntegral(a): RngOrdElt -> BoolElt`

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): FldAlgElt -> BoolElt`
`IsPrimitive(a): RngOrdElt -> BoolElt`

Returns `true` if the element a of the algebraic field F or one of its orders O generates F.

`IsTorsionUnit(w): RngOrdElt -> BoolElt`

Returns `true` if and only if the order element w is a unit of finite order.

`IsPower(w, n): RngOrdElt, RngIntElt -> BoolElt, RngOrdElt`

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): RngOrdElt -> BoolElt`
`IsTotallyPositive(a): FldAlgElt -> BoolElt`

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): FldAlgElt -> BoolElt`

`IsZero(w): RngOrdElt -> BoolElt`

`IsOne(a): FldAlgElt -> BoolElt`

`IsOne(w): RngOrdElt -> BoolElt`

`IsMinusOne(a): FldAlgElt -> BoolElt`

`IsMinusOne(w): RngOrdElt -> BoolElt`

`IsUnit(a): FldAlgElt -> BoolElt`

`IsUnit(w): RngOrdElt -> BoolElt`

`IsNilpotent(a): FldAlgElt -> BoolElt`

`IsNilpotent(a): RngOrdElt -> BoolElt`

`IsIdempotent(a): FldAlgElt -> BoolElt`

`IsIdempotent(a): RngOrdElt -> BoolElt`

`IsZeroDivisor(a): FldAlgElt -> BoolElt`

`IsZeroDivisor(a): RngOrdElt -> BoolElt`

`IsRegular(a): FldAlgElt -> BoolElt`

`IsRegular(a): RngOrdElt -> BoolElt`

`IsIrreducible(a): FldAlgElt -> BoolElt`

`IsIrreducible(a): RngOrdElt -> BoolElt`

`IsPrime(a): FldAlgElt -> BoolElt`

`IsPrime(a): RngOrdElt -> BoolElt`

39.7.5 Field Generators

`K . 1: FldNum -> FldNumElt`

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): FldNum -> FldNumElt`

`PrimitiveElement(F): FldOrd -> FldOrdElt`

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.

`PrimitiveElement(O): RngOrd -> RngOrdElt`

Given an order O, returns a primitive element for `FieldOfFractions(O)`.

`Generators(K): FldAlg -> [FldAlgElt]`

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): FldAlg, FldAlg -> [FldAlgElt]`

`Generators(K, Q): FldAlg, FldRat -> [FldAlgElt]`

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.

39.7.6 Real and Complex Embeddings

The precision of the real and complex numbers returned by the functions below should be controlled using the optional argument `Precision` in each function. *Note:* `SetKantPrecision` should not be used for this purpose.

`Conjugates(a): FldAlgElt -> [ FldComElt ]`

`Conjugates(a): RngOrdElt -> [ FldComElt ]`

Precision: RngIntElt	Default:

Given an element a in a number field or order, this returns a sequence of complex numbers, which are the real and complex embeddings of a.

When a is an element of an *absolute* field or order R (i.e. R has base field $\mathbb{Q}$ or $\mathbb{Z}$), the number of embeddings is the degree of R. The r_1 real conjugates are listed first, followed by the r_2 pairs of complex conjugates.

When a is an element of a *relative* field or order R, the sequence contains `Evaluate(a, v)` where v runs through `InfinitePlaces(R)`. In particular, it contains only $r_1 + r_2$ numbers.

In both cases, the conjugates are given in a fixed ordering which depends only on (the defining polynomials of) the field or order.

`Conjugate(a, l): FldAlgElt, RngIntElt -> FldReElt`

`Conjugate(a, l): RngOrdElt, RngIntElt -> FldReElt`

Precision: RngIntElt	Default:

For a in an *absolute* field or order, this returns `Conjugates(a)[l]`.

`Conjugate(a, l): FldAlgElt, [RngIntElt] -> FldReElt`

Precision: RngIntElt	Default:

For a in a *relative* field or order, this returns the conjugate of a indexed by l, where $l = [l_1, \ldots, l_n]$ is a sequence of integers.

Let K be the parent of a, defined as a tower of extensions with n steps $\mathbb{Q} \subseteq K_1 \subseteq \ldots \subseteq K_n = K$. The embedding indexed by l is defined inductively: it extends the embedding $[l_1, \ldots, l_{n-1}]$ of K_{n-1} and is the l_nth such extension. These extensions are ordered by a fixed rule, depending only on (the defining polynomials of) the field tower.

The `InfinitePlaces` of K are indexed the same way.

`AbsoluteValues(a): FldAlgElt -> [FldReElt]`

`AbsoluteValues(a): RngOrdElt -> [FldReElt]`

This returns a sequence of $r_1 + r_2$ positive real numbers. These are the absolute values of the real and complex embeddings of the element a, which is required to be an element of an absolute field or order.

`Logs(a): FldAlgElt -> [FldReElt]`

`Logs(a): RngOrdElt -> [FldReElt]`

This returns a sequence of $r_1 + r_2$ positive real numbers. These are the natural logs of the absolute values of the real and complex embeddings of the element a, which is required to be a nonzero element of an absolute field or order.

`InfinitePlaces(K): FldAlg -> [PlcNumElt]`

`InfinitePlaces(O): RngOrd -> [PlcNumElt]`

This returns a sequence containing all the infinite places of the field. Each place corresponds to a real embedding or a pair of complex embeddings. The ordering of the places is fixed.

`Evaluate(x, p): FldAlgElt, PlcNumElt -> RngElt`

`Evaluate(x, p): RngOrdElt, PlcNumElt -> RngElt`

When p is a finite place, this returns the image of x in the residue class field corresponding to p.

When p is an infinite place, this returns the image of x under the corresponding embedding, i.e., a real or complex number. (This can also be obtained using `Conjugate` or `Conjugates`.)

`RealEmbeddings(a): FldAlgElt -> []`

`RealEmbeddings(a): RngOrdElt -> []`

This returns the sequence of values `Evaluate(a,p)` where p runs through the real places of the field.

`MinkowskiLattice(O): RngOrd -> Lat, Map`

`Lattice(O): RngOrd -> Lat, Map`

`Precision: RngIntElt` `Default:`

Given an absolute order O, returns the lattice determined by the real and complex embeddings of O.

`MinkowskiLattice(I): RngOrdIdl -> Lat, Map`

`Lattice(I): RngOrdIdl -> Lat, Map`

`Precision: RngIntElt` `Default:`

Given an ideal I in an absolute order, returns the lattice determined by the real and complex embeddings of I.

`MinkowskiSpace(F): FldAlg -> Lat, Map`

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 $F \to V$.

39.7.7 Heights

The real precision of results can be controlled using `SetKantPrecision`.

`AbsoluteLogarithmicHeight(a): FldAlgElt -> FldReElt`

`AbsoluteLogarithmicHeight(a): RngOrdElt -> FldReElt`

Let P be the minimal polynomial of the element a over $\mathbb{Z}$, with leading coefficient a_0 and roots $\alpha_1, \ldots, \alpha_n$. Then the absolute logarithmic height is defined to be

$$h(\alpha) = \frac{1}{n} \log(a_0 \prod_{j=1}^{n} \max(1, |\alpha_j|)).$$

`CoefficientHeight(E): RngOrdElt -> RngIntElt`

`CoefficientHeight(E): FldAlgElt -> RngIntElt`

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): RngOrdElt -> RngIntElt`

`CoefficientLength(E): FldAlgElt -> RngIntElt`

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.

`Length(a): FldAlgElt -> FldReElt`

`Length(a): RngOrdElt -> FldReElt`

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.

Example: Alternative Discriminant Function (ex-19f6ce)

It is not hard to write an alternative discriminant function, using the `Conjugates` of the basis `[0.1, 0.2, ...]`.

```
> function disc(O)
>     B := [ Conjugates(O.i) : i in [1 .. Degree(O)] ];
>     D := Determinant(Matrix(B))^2;
>     return RealField(20) ! D;
> end function;
> _<x> := PolynomialRing(Integers());
> O := MaximalOrder(NumberField(x^4 - 420*x^2 + 40000));
> disc(O);
42025.000000000000001
> Discriminant(O);
42025
```

The function `disc` obtains a real approximation to the exact value given by `Discriminant`. 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))] ] >;
```

39.7.8 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 $\mathbb{Z}$ or $\mathbb{Q}$.

`Norm(a): FldAlgElt -> FldAlgElt`

`Norm(a): FldAlgElt -> FldRatElt`

`Norm(a, R): FldAlgElt, Rng -> RngElt`

`Norm(a): RngOrdElt -> RngOrdElt`

`Norm(a): RngOrdElt -> RngIntElt`

`Norm(a, R): RngOrdElt, Rng -> RngElt`

The relative norm $\mathrm{N}_{\mathrm{L/F}}(\mathrm{a})$ 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): FldAlgElt -> FldRatElt`

`NormAbs(a): FldAlgElt -> FldRatElt`

`AbsoluteNorm(a): RngOrdElt -> FldRatElt`

`NormAbs(a): RngOrdElt -> FldRatElt`

The absolute norm $\mathrm{N}_{\mathrm{L}/\mathbb{Q}}(\mathrm{a})$ over $\mathbb{Q}$ of the element a of L (or one of its orders).

`Trace(a): FldAlgElt -> FldAlgElt`

`Trace(a): FldAlgElt -> FldRatElt`

`Trace(a, R): FldAlgElt, Rng -> RngElt`

`Trace(a): RngOrdElt -> RngOrdElt`

`Trace(a): RngOrdElt -> RngIntElt`

`Trace(a, R): RngOrdElt, Rng -> RngElt`

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): FldAlgElt -> FldRatElt`

`TraceAbs(a): FldAlgElt -> FldRatElt`

`AbsoluteTrace(a): RngOrdElt -> FldRatElt`

`TraceAbs(a): RngOrdElt -> FldRatElt`

The absolute trace $\mathrm{Tr}_{L/\mathbb{Q}}(a)$ over $\mathbb{Q}$ of the element a of L (or one of its orders).

`CharacteristicPolynomial(a): FldAlgElt -> RngUPolElt`

`CharacteristicPolynomial(a, R): FldAlgElt, Rng -> RngUPolElt`

`CharacteristicPolynomial(a): RngOrdElt -> RngUPolElt`

`CharacteristicPolynomial(a, R): RngOrdElt, Rng -> RngUPolElt`

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): FldAlgElt -> RngUPolElt`

`AbsoluteCharacteristicPolynomial(a): RngOrdElt -> RngUPolElt`

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): FldAlgElt -> RngUPolElt`

`MinimalPolynomial(a, R): FldAlgElt, Rng -> RngUPolElt`

`MinimalPolynomial(a): RngOrdElt -> RngUPolElt`

`MinimalPolynomial(a, R): RngOrdElt, Rng -> RngUPolElt`

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): FldAlgElt -> RngUPolElt`

`AbsoluteMinimalPolynomial(a): RngOrdElt -> RngUPolElt`

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): FldAlgElt -> AlgMatElt`

`RepresentationMatrix(a, R): FldAlgElt, Rng -> AlgMatElt`

`RepresentationMatrix(a): RngOrdElt -> AlgMatElt`

`RepresentationMatrix(a, R): RngOrdElt, Rng -> AlgMatElt`

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): FldAlgElt -> AlgMatElt`

`AbsoluteRepresentationMatrix(a): RngOrdElt -> AlgMatElt`

Return the representation matrix of a relative to the $\mathbb{Q}$-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 $L_0 := \mathbb{Q}$. Then the representation matrix is computed with respect to the $\mathbb{Q}$-basis $(\prod_j \omega_{i_j,j})_{i\in I}$ consisting of products of basis elements of the different levels.

Example: Norm, Trace, and Minimal Polynomial `(ex-68a043)`

We create the norm, trace, minimal polynomial, and representation matrix of the element $\alpha/2$ in the quartic field $\mathbb{Q}(\alpha)$.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^4-420*x^2+40000);
> z := y/2;
```

(continues on next page)

(continued from previous page)

```
> Norm(z), Trace(z);
2500 0
> MinimalPolynomial(z);
$.1^4 - 105*$.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
```

39.7.9 The Quadratic Defect

Let K be a number field or the field of rationals and let p be a prime ideal in the ring of integers O of K. Further, let K_p be the completion of K at p. Every square class of $K_p^*/(K_p^*)^2$ has a representative in K. The following functions allow to work with these classes, without the construction of the completion K_p explicitly.

`QuadraticDefect(a, p): RngElt, RngOrdIdl -> RngIntElt`

`QuadraticDefect(a, p): FldRatElt, RngIntElt -> RngIntElt`

`QuadraticDefect(a, p): RngIntElt, RngIntElt -> RngIntElt`

`QuadraticDefect(a, p): FldRatElt, RngInt -> RngIntElt`

`QuadraticDefect(a, p): RngIntElt, RngInt -> RngIntElt`

`RelativeQuadraticDefect(a, p): RngElt, RngOrdIdl -> RngIntElt`

`RelativeQuadraticDefect(a, p): FldRatElt, RngIntElt -> RngIntElt`

`RelativeQuadraticDefect(a, p): RngIntElt, RngIntElt -> RngIntElt`

`RelativeQuadraticDefect(a, p): FldRatElt, RngInt -> RngIntElt`

`RelativeQuadraticDefect(a, p): RngIntElt, RngInt -> RngIntElt`

For an element a in K, let

$$q_p(a) := \bigcap_{b \in K_p} (a - b^2)O_p$$

be the quadratic defect of a in the completion K_p. Further $q_p(a)/a$ is called the relative quadratic defect of $a \neq 0$ in K_p.

The above function computes the valuation of the (relative) quadratic defect of a in K_p. Note that the valuation might be infinity, which happens if and only if a is a square in K_p.

`IsLocalSquare(a, p): RngElt, RngOrdIdl -> BoolElt`

`IsLocalSquare(a, p): FldRatElt, RngIntElt -> BoolElt`

`IsLocalSquare(a, p): RngIntElt, RngIntElt -> BoolElt`

`IsLocalSquare(a, p): FldRatElt, RngInt -> BoolElt`

`IsLocalSquare(a, p): RngIntElt, RngInt -> BoolElt`

Given an element a in K, decide if a is a square in the completion K_p.

`LocalMultiplicativeGroupModSquares(p): RngOrdIdl -> ModFld, Map`

`LocalMultiplicativeGroupModSquares(p): RngIntElt -> ModFld, Map`

Let K be a number field or the rationals and let p be a prime ideal of the ring of integers of K. Then the units of the completion modulo squares $K_p^*/(K_p^*)^2$ form a $GF(2)$-vector space V. This function returns V and a map from K to V which can compute inverses.

If V has rank r say, then the first $r-1$ basis vectors of V generate the subspace corresponding to the elements of $O_p^*/(O_p^*)^2$.

`UnitSquareClassReps(p): RngOrdIdl -> SeqEnum`

`UnitSquareClassReps(p): RngIntElt -> SeqEnum`

Returns a sequence of elements in O that represent the elements of $O_p^*/(O_p^*)^2$.

`NiceUnitSquareClassRepresentative(u, p): RngElt, RngOrdIdl -> RngElt`

`NiceUnitSquareClassRepresentative(u, p): RngElt, RngIntElt -> RngElt`

Given an element in u in K which has p-adic valuation 0, return some element v in K such that uv is a square in O_p^* and the quadratic defect of u is $(v-1)O_p$.

39.7.10 Other Functions

`ElementToSequence(a): FldAlgElt -> [ FldAlgElt ]`

`Eltseq(a): FldAlgElt -> [ FldAlgElt ]`

`ElementToSequence(a): RngOrdElt -> [ FldOrdElt ]`

`Eltseq(a): RngOrdElt -> [ FldOrdElt ]`

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): FldAlgElt, FldAlg -> [RngElt]`

`Eltseq(E, k): FldAlgElt, Rng -> [RngElt]`

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): FldAlgElt -> [FldRatElt]`

Given an element in a field K, this returns the coordinates of e with respect to the `AbsoluteBasis` of K over $\mathbb{Q}$.

`a[i]: FldAlgElt, RngIntElt -> FldRatElt`

`a[i]: FldAlgElt, RngIntElt -> FldAlgElt`

`a[i]: RngOrdElt, RngIntElt -> FldRatElt`

`a[i]: RngOrdElt, RngIntElt -> FldOrdElt`

The coefficient of the ith basis element in the algebraic field or order element a.

`ProductRepresentation(a): RngOrdElt -> [ RngOrdElt ], [ RngIntElt ]`

`ProductRepresentation(a): FldAlgElt -> [ FldAlgElt ], [ RngIntElt ]`

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): [ FldAlgElt ], [ RngIntElt ] -> FldAlgElt`

`PowerProduct(P, E): [FldAlgElt], [RngIntElt] -> FldAlgElt`

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): RngOrdElt, RngOrdIdl -> RngIntElt`

`Valuation(w, I): FldAlgElt, RngOrdIdl -> RngIntElt`

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 *Ideals and Quotients*.

`Decomposition(a): RngOrdElt -> SeqEnum[<RngOrdIdl, RngIntElt>]`

`Decomposition(a): FldOrdElt -> SeqEnum[<RngOrdIdl, RngIntElt>]`

The factorization of the order or algebraic field element a into prime ideals.

`Divisors(a): RngOrdElt -> SeqEnum[RngOrdElt]`

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): RngOrdElt -> RngIntElt`

The index of the module $\mathbb{Z}[a]$ in O where a lies in O, an order over $\mathbb{Z}$. If a is not a primitive element the index is infinite.

`Different(a): RngOrdElt -> RngOrdElt`

The different of the element a of an order of a number field.

`DegreeOnePrimeIdeals(O, B): RngOrd, RngIntElt -> [ RngOrdIdl ]`

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.

39.8 Ideal Class Groups

This section describes functions for computing the group of ideal classes of the maximal order of an absolute number field. The method usually employed is the *relation* method ([Coh93, Hes96]), 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 rigorously, one must ensure 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. It should be stressed that, by default, a guaranteed result is computed using the Minkowski bound. Even for innocent looking fields, this may take considerable time. In contrast, Pari (from version 2.0), by default, uses a much smaller bound giving results that are not guaranteed. In Magma, to perform computations comparable to Pari, the user must request a non-rigorous computation. The recommended way is to globally control this by using `SetClassGroupBounds("GRH")`, before calling routines involving class group calculations. Note that it is better to give `"GRH"` here than to give a numerical value for the bound. Starting from version 2.20, a new implementation of the sieve algorithm for finding relations ([Bia]) is used automatically, as a subroutine, to the extent it is advantageous. This method is very effective for fields of degree up to 5, and to a lesser extent degree 6. When a class group computation has been completed, the results are stored with the order (to avoid repeated computation).

`SetPrintClassGroupWarnings(b): BoolElt`

Set whether the class group algorithm prints warnings about expensive computations.

`ClassGroup(O: parameters): RngOrd -> GrpAb, Map`

`ClassGroup(K: parameters): FldNum -> GrpAb, Map`

```
Bound          : RngIntElt                    Default: MinkowskiBound
Proof          : MonStgElt                    Default: "Full"
Al             : MonStgElt                    Default: "Automatic"
UsePowerProduct: BoolElt                      Default: false
```

(continues on next page)

(continued from previous page)

```
ClassGroup     : Verbose                          Default: 3 % internally 5, but > 3␣
↪is not useful for users
```

The group of ideal classes in the ring of integers O of a number field K is returned as an abelian group, together with a map from this abstract group to the set of ideal of O. The map is defined in both directions: in the inverse direction, it returns the ideal class of a given ideal of O. Since V2.28, a distributed parallel version of the main algorithm is available. As usual, one first uses `SetNthreads` and optionally `StartWorkers` to select the number of cores/workers before the class group algorithm is called. By default, all computations are unconditionally rigorous: this means that a final step must be performed, checking all prime ideals up to the Minkowski bound (which is impractical for many fields). To perform conditional computations (under "GRH", or with a smaller bound), the recommended approach is to set the rigour for all class groups by using `SetClassGroupBounds("GRH")` before calling `ClassGroup`. Note that it is better to give `"GRH"` than to give a numerical value for the bound. The alternative approach is to use the parameters `Proof` and `Bound`, which have the following effect. If `Proof := "GRH"`, a bound for generators of the class group based on the GRH is used to replace the Minkowski bound. Further, a GRH based aopproximation of the residue of the Dedekind zeta function is assumed to be correct. The result will hence be correct under the GRH. 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 := "Bound"`, the maximum of the given bound and a GRH based bound for a generating set of the class group is used. Further, a GRH based aopproximation of the residue of the Dedekind zeta function is assumed to be correct. The result will hence be correct under the GRH. 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) the bound given by `"SetClassGroupBounds"` is used. By default this is the Minkowski bound and a guaranteed result is computed. This is equivalent to `Bound := MinkowskiBound(K)` and `Proof := "Subgroup"`. But, if `SetClassGroupBounds("GRH");` was used, the result is conditional and correct under the GRH.

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.

For quadratic fields, alternative algorithms may be selected using the `Al` parameter.

In some previous versions of Magma, use of the sieve algorithm was controlled by setting `Al` to `"Sieve"` or to `"NoSieve"`. This usage is deprecated, as the choice is now made internally: the sieve algorithm is called automatically by the main class group routine to the extent it is advantageous.

If `UsePowerProduct` is set to `true` then images of group elements under the map from the class group will be returned as a power product of ideals.

`RingClassGroup(O): RngOrd -> GrpAb, Map`

`PicardGroup(O): RngOrd -> GrpAb, Map`

```
UsePowerProduct: BoolElt                    Default: false
```

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 principal ideals. The algorithm and its implementation are due to Klüners and Pauli, [PK05].

If `UsePowerProduct` is set to `true`, any computations using the images of elements of the class group will not evaluate a power product representation of ideals but instead work with this representation. This can be substantially more efficient for some orders.

`ConditionalClassGroup(O): RngOrd -> GrpAb, Map`

ConditionalClassGroup(K): FldNum -> GrpAb, Map

This is equivalent to calling ClassGroup(O:Proof := "GRH").

ClassGroupPrimeRepresentatives(O, I): RngOrd, RngOrdIdl -> Map

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.

PrimeRepresentative(c, f: parameters): GrpAbElt, Map -> RngOrdIdl

```
DegreeOne: BoolElt                    Default: false
```

Given an element c of the abelian group A and the map f returned by ClassGroup for a maximal order O, this function returns a prime ideal of O lying in the ideal class c. If DegreeOne is set to true, the prime ideal returned will have residue degree one over the rationals. If Minimal is set to true, the search proceeds by splitting successive rational primes and testing whether a prime above lies in the class c, which tends to produce an ideal of small norm. Otherwise, the search enumerates lattice points of O until an ideal in the required class is found.

PrimeRepresentative(I: parameters): RngOrdIdl -> RngOrdIdl

```
DegreeOne: BoolElt                    Default: false
Minimal  : BoolElt                    Default: false
```

Given an ideal I of a maximal order O of a number field, this function returns a prime ideal of O lying in the same ideal class as I. The search is carried out by enumerating lattice points of O. If DegreeOne is set to true, the prime ideal returned will have residue degree one.

ClassNumber(O: parameters): RngOrd -> RngIntElt

ClassNumber(K: parameters): FldNum -> RngIntElt

```
Bound     : RngIntElt                 Default: MinkowskiBound
Proof     : MonStgElt                 Default: "Full"
Al        : MonStgElt                 Default: "Automatic"
ClassGroup: Verbose                   Default: 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.

MinkowskiBound(K): FldNum -> RngIntElt

MinkowskiBound(O): RngOrd -> RngIntElt

This returns the Minkowski bound for the maximal order of the given field K. This is an unconditional integer upper bound for norms of the generators of the ideal class group of the maximal order.

BachBound(K): FldNum -> RngIntElt

BachBound(O): RngOrd -> RngIntElt

This returns the Bach bound for the maximal order of the given field K. This is an integer upper bound for norms of the generators of the ideal class group of the maximal order which holds if the generalized Riemann hypothesis is true.

GRHBound(K): FldNum -> RngIntElt

GRHBound(O): RngOrd -> RngIntElt

This returns an integer upper bound, proven assuming the generalized Riemann hypothesis, for norms of the generators of the ideal class group of the maximal order of the given field K. The function returns the best bound obtainable in Magma. In the current version, in addition to the Bach bound, the algorithm uses results of Belabas and Grenie, Molteni, which often produce bounds that are significantly smaller than the Bach bound.

`FactorBasisVerify(O, a, b): RngOrd, RngIntElt, RngIntElt`

This verifies that every prime ideal of the order O with norm between a and b is equivalent to an ideal of O with norm smaller than a. This function requires that a class group computation has already been done for O.

39.8.1 Class Group Internals

The results of a class group computation, which are stored internally on the order O, are a factor base, a set of relations, and the relation matrix. (Units are also stored, and used in later computations.) Access to this data is provided by functions below. Also listed are functions for independently producing data of the same form; these do not use the same code as the internal class group computation.

`EulerProduct(O, B): RngOrd, RngIntElt -> FldReElt`

Computes an approximation of the residue of the Dedekind zeta function of O at $s = 1$ via the Euler product for the order O using only prime ideals over prime numbers of norm $\leq B$.

`ResidueGRH(O, B): RngOrd, RngIntElt -> FldReElt`

Computes an approximation of the residue of the Dedekind zeta function of O at $s = 1$ as a sum involving only prime ideals of norm $\leq B$. Assuming GRH, the best known error bounds for `ResidueGRH` are smaller than the error bounds for `EulerProduct`.

`ResidueGRHbound(O, e): RngOrd, FldReElt -> FldReElt`

Assuming GRH, this function computes a bound B such that `ResidueGRH` will have a logarithmic error less than e [BF15].

`FactorBasis(K, B): FldNum, RngIntElt -> [ RngOrdIdl ]`

`FactorBasis(O, B): RngOrd, RngIntElt -> [ RngOrdIdl ]`

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): RngOrd -> [ RngOrdIdl ], Integer`

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(O): RngOrd -> ModHomElt`

Given a maximal order O where the class group has been computed previously, the resulting relation matrix is returned.

`Relations(O): RngOrd -> ModHomElt`

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): RngOrd -> ModHomElt`

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: Class Group of a Real Quadratic Field `(ex-92eeba)`

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
```

(continues on next page)

(continued from previous page)

```
> 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;
O
> IsPrincipal(p);
true
> p := Decomposition(O, 37)[1][1];
> p @@ m;
C.1
> IsPrincipal(p);
false
> MinkowskiBound(O);
3
> F, B := FactorBasis(O);
> B;
50
```

Note that although the `MinkowskiBound` is only 3, the algorithm chose a larger bound for the computation internally.

```
> 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)
```

The `RelationMatrix` stores the valuation of the `Relations` at each prime ideal contained in the `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
> GRHBound(K);
```

(continues on next page)

(continued from previous page)

```
259
> time C := ClassGroup(K);
Time: 0.510
> C;
Abelian Group isomorphic to Z/10
Defined on 1 generator
Relations:
    10*$.1 = 0
```

This is an unconditional result. The conditional computation, assuming GRH, is faster. Note that the `BachBound` should NOT be used – the better `GRHBound` is much smaller. (They both are bounds which guarantee the class group computation is correct assuming GRH.)

```
> K := NumberField(x^5-14*x^4+14*x^3-14*x^2+14*x-14);
> time #ClassGroup(K : Proof := "GRH");
10
Time: 0.100
> K := NumberField(x^5-14*x^4+14*x^3-14*x^2+14*x-14);
> time #ClassGroup(K : Bound := BachBound(K));
10
Time: 0.280
```

39.8.2 Setting the Class Group Bounds

These functions control the rigour of all subsequent of class group computations. The recommended way is `SetClassGroupBounds("GRH")`. This is the minimum possible level of rigour with the current implementation – all class groups will be correct if GRH holds, regardless of all settings.

`SetClassGroupBounds(string): MonStgElt`

If this is called with the string "GRH", all subsequent class group computations will be performed in such a way that either they are correct, or GRH does not hold.

`SetClassGroupBounds(n): RngIntElt`

The integer n is the proof bound to be used in all subsequent calls to `ClassGroup`. That is, each class group computed will be correct if it is generated by ideals of norm at most n.

Example: Setting Class Group Bounds `(ex-3fbfbd)`

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);
```

39.8.3 Class Group Map Caching

The map returned by `ClassGroup` can be used to compute the ideal class of a given ideal. In applications which involve many repeated calls to this, it may be advantageous to store the results of each call (although this may also use a lot of memory). By default, results are not stored.

For example, if the order O is to be used extensively as a coefficient ring for class field computations, then every time discrete logarithms 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.

`ClassGroupGetUseMemory(O): RngOrd -> BoolElt`

`ClassGroupSetUseMemory(O, f): RngOrd, BoolElt`

These functions access and set whether or not the map returned by `ClassGroup(O)` stores the results of calls to the inverse map.

39.9 Unit Groups

For most number fields, the algorithm used to compute the unit group involves computing the class group and obtaining units from the 'class group relations'. If a class group has already been computed for the field, these 'unit relations' are already stored; otherwise, the standard class group computation will be performed first.

The unit group routine is fully rigorous, even when the class group computation is not. For details see *UnitGroup* and `IndependentUnits` below.

Previously, some of the algorithms described in [PZ89], (pp. 343–344), and in [Poh93] were used but since V2.21 the class group, computed assuming GRH if not already computed, is used to compute the independent units.

`UnitRank(O): RngOrd -> RngIntElt`

`UnitRank(K): FldAlg -> RngIntElt`

The free rank of the group of units in the ring of integers O (of a number field K).

`TorsionUnitGroup(O): RngOrd -> GrpAb, Map`

`TorsionUnitGroup(K): FldNum -> GrpAb, Map`

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)`.

`UnitGroup(O): RngOrd -> GrpAb, Map`

`MultiplicativeGroup(O): RngOrd -> GrpAb, Map`

```
Al       : MonStgElt                    Default: "Automatic"
GRH      : BoolElt                      Default: false
UnitGroup: Verbose                      Default: 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 $2 \leq i \leq r_1 + r_2$. By default, `UnitGroup` performs a proof phase, to verify rigorously that the unit group returned is the full unit group rather than a finite index subgroup. This is done even when the class group computation was only heuristic. It is done by a different method than the proof phase for the class group (and it does not prove the class group). When `UnitGroup` is called with the option `GRH` set to `true`, this proof phase is skipped, which means the calculation has the same level of rigour as the class group computation. This is equivalent to doing the following: first compute a class group, specifying the rigour of that computation (see *Ideal Class Groups*), and then call `IndependentUnits` (see below).

`UnitGroup(K): FldNum -> GrpAb, Map`

`MultiplicativeGroup(K): FldNum -> GrpAb, Map`

This is identical to `UnitGroup(MaximalOrder(K))`.

`IndependentUnits(O): RngOrd -> GrpAb, Map`

`IndependentUnits(K): FldNum -> GrpAb, Map`

```
Al       : MonStgElt                    Default: "Automatic"
UnitGroup: Verbose                      Default: 6
```

This function is an alternative to `UnitGroup` which returns a subgroup of finite index in the unit group. The arguments and return values are the same as for `UnitGroup`. In the current implementation, `IndependentUnits` returns the full unit group whenever the class group computation is correct. This is always the case if the GRH holds. After calling `IndependentUnits(O)`, if *SetOrderUnitsAreFundamental*`(O)` is invoked Magma will assume this unit group is the full unit group in all subsequent calculations with the same number field.

`pFundamentalUnits(O, p): RngOrd, RngIntElt -> GrpAb, Map`

`pFundamentalUnits(K, p): FldNum, RngIntElt -> GrpAb, Map`

```
Al       : MonStgElt                    Default: "Automatic"
UnitGroup: Verbose                      Default: 6
```

Given a maximal 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.

`UnitGroupAsSubgroup(O): RngOrd -> GrpAb`

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 implementation is due to Klüners and Pauli, [PK05].

`MergeUnits(K, a): FldNum, FldNumElt -> BoolElt`

`MergeUnits(O, a): RngOrd, RngOrdElt -> BoolElt`

```
UnitGroup: Verbose                    Default: 6
```

This function is largely irrelevant in the current implementation.

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.

Example: Class and Unit Groups of a Quartic Field (ex-98001b)

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): RngOrdElt -> BoolElt`

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): RngOrdElt -> [RngOrdElt]

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): RngOrd -> [RngOrdElt]

```
UnitEq: Verbose                    Default: 5
```

This function returns a sequence S of exceptional units of the order O.

UnitsWithSigns(O, oo, Signs): RngOrd, [PlcNumElt], [RngInt] -> [RngOrdElt]

UnitsWithSigns(K, oo, Signs): FldAlg, [PlcNumElt], [RngInt] -> [FldAlgElt]

Returns all units in O (or the ring of integers of K) modulo squares that have sign $Signs[i]$ at the i-th real place given in $oo[i]$.

UnitsWithSigns(O, x): RngOrd, RngElt -> [RngOrdElt]

UnitsWithSigns(K, x): FldAlg, RngElt -> [FldAlgElt]

Returns all units in O (or the ring of integers of K) modulo squares that have the same signs at the real places of O as the element x.

UnitsWithSigns(x): RngOrdElt -> [RngOrdElt]

UnitsWithSigns(x): FldAlgElt -> [FldAlgElt]

Given an element x in an order O or a number field K, return all units of O (or the ring of integers of K) modulo squares that have the same signs at the real places of O as x.

HasTotallyPositiveGenerator(I): RngOrdFracIdl -> BoolElt, [RngOrdElt]

Decides if I is generated by some totally positive element. If so, the second return value contains all such generators up to squares.

TraceMinimalCone(nf): FldNum -> TorCon

Given a totally real number field, find the trace-minimal cone, which is a fundamental domain for the action of the totally positive units.

39.10 Diophantine Equations

Magma has routines for solving norm equations, Thue equations, index form equations, and unit equations.

39.10.1 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): RngOrd, RngIntElt -> BoolElt, [RngOrdElt]

`NormEquation(O, m): RngOrd, RngOrdElt -> BoolElt, [ RngOrdElt ]`

```
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 $\mathrm{N}_{\mathrm{F/L}}(\alpha)$, 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 ([FJP97, Fie97, Jur93]).

`NormEquation(F, m): FldAlg, RngIntElt -> BoolElt, [ FldAlgElt ]`

`NormEquation(F, m): FldAlg, FldAlgElt -> BoolElt, [ FldAlgElt ]`

`NormEquation(F, m): FldAlg, FldRatElt -> BoolElt, [ FldAlgElt ]`

```
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, Gar80, Sim02]. 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 $m\mathbb{Z}_F$. 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): RngElt, Map -> BoolElt, RngElt`

```
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): RngElt, Map, RngOrd -> BoolElt, [RngOrdElt]

```
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): FldNum, FldNumElt, FldNumElt -> BoolElt, [FldNumElt]

```
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: Solving a Relative Norm Equation (ex-a68dc5)

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;
```

39.10.2 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 system (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.

`ThueSystem(f): RngUPolElt -> Thue`

Given a polynomial f of degree at least 2 over the integers, this function returns the 'Thue system' corresponding to f; such systems are used by the functions for solving Thue equations. They are printed as the homogeneous version of f.

`ThueSystem(O): RngOrd -> Thue`

Given an order O with $\mathbb{Z}$ as its coefficient ring, this function returns the Thue system corresponding to the defining polynomial of O.

`Evaluate(t, a, b): Thue, RngIntElt, RngIntElt -> RngIntElt`

`Evaluate(t, S): Thue, [ RngIntElt ] -> RngIntElt`

Given a Thue system 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): Thue, RngIntElt -> [ [ RngIntElt, RngIntElt ] ]`

```
Exact : BoolElt                    Default: true
ThueEq: Verbose                    Default: 5
```

Given a Thue system 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: Solving a Thue Equation (ex-cbfc14)

A use of thue equations is shown.

```
> R<x> := PolynomialRing(Integers());
> f := x^3 + x + 1;
> T := ThueSystem(f);
> T;
Thue system with form:  X^3 + X Y^2 + Y^3
> Evaluate(T, 3, 2);
47
> Solutions(T, 4);
```

(continues on next page)

(continued from previous page)

```
[]
> 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 ]
```

39.10.3 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): FldNumElt, FldNumElt, FldNumElt -> [ ModHomElt ]`

```
UnitEq: Verbose                    Default: 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: Solving a Unit Equation `(ex-5cb066)`

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]]
]
```

39.10.4 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): RngOrd, RngIntElt -> [ RngOrdElt ]`

```
IndexFormEquation: Verbose                    Default: 5
```

Given an absolute order O, this function will find "all" (up to equivalence) solutions $\alpha \in O$ to $(O : Z[\alpha]) = k$.

In the current implementation, the unit rank must be no more than 10.

Example: Integral Power Bases via Index Forms `(ex-a3384f)`

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
```

39.11 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 $\mathbb{Z}$ 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. Fractional ideals over $\mathbb{Q}$ are of type `RngIntFracIdl`, while integral ideals in $\mathbb{Z}$ have the type `RngInt`.

39.11.1 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: RngElt, RngOrd -> RngOrdFracIdl`

`O * x: RngElt, RngOrd -> RngOrdFracIdl`

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: RngOrd, RngInt -> RngOrdFracIdl`

`F !! I: FldOrd, RngOrdFracIdl -> RngOrdFracIdl`

`F !! I: FldRat, RngOrdFracIdl -> RngIntFracIdl`

`F !! I: FldRat, RngIntFracIdl -> RngIntFracIdl`

`F !! I: FldOrd, RngIntFracIdl -> RngOrdFracIdl`

`O !! I: RngOrd, RngOrdFracIdl -> RngOrdIdl`

`O !! I: RngInt, RngInt -> RngInt`

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 >: RngOrd, RngElt, ..., RngElt -> RngOrdFracIdl`

`ideal<O | x>: RngOrd, RngIntElt -> RngOrdIdl`

`ideal<O | M, d>: RngOrd, AlgMatElt, RngIntElt -> RngOrdFracIdl`

`ideal<O | M, d>: RngOrd, ModDed, RngIntElt -> RngOrdFracIdl`

`ideal<O | M, I1, ..., In>: RngOrd, AlgMatElt, RngOrdFracIdl, ..., RngOrdFracIdl -> RngOrdFracIdl`

`ideal<O | M, [I1, ..., In]>: RngOrd, AlgMatElt, [RngOrdFracIdl] -> RngOrdFracIdl`

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: Ideal Generated by an Element (ex-dc3704)

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, 1, 1, 0]
> FieldOfFractions(O)!!I;
Principal Ideal of O
Generator:
    0.2 + 0.3
> O!!$1 eq I;
true
```

It is also possible to construct fractional ideals in $\mathbb{Q}$.

`FractionalIdeal(x): FldRatElt -> RngIntFracIdl`

`FractionalIdeal(x): RngIntElt -> RngIntFracIdl`

`FractionalIdeal(x): [ FldRatElt ] -> RngIntFracIdl`

`FractionalIdeal(x): [ RngIntElt ] -> RngIntFracIdl`

The fractional ideal generated by x.

`FractionalIdeal(I): RngInt -> RngIntFracIdl`

Make I into a fractional ideal.

39.11.2 Invariants

Some information describing an ideal can be retrieved.

`Order(I): RngOrdFracIdl -> RngOrd`

`Order(I): RngIntFracIdl -> RngInt`

The order O which the ideal I is of.

`Denominator(I): RngOrdFracIdl -> RngIntElt`

`Denominator(I): RngIntFracIdl -> RngIntElt`

The denominator of the fractional ideal I. This is the smallest positive integer d such that dI is an integral ideal.

`PrimitiveElement(I): RngOrdIdl -> RngOrdElt`

`PrimitiveElement(I): RngOrdFracIdl -> FldOrdElt`

`UniformizingElement(P): RngOrdIdl -> RngOrdElt`

`UniformizingElement(P): RngInt -> RngIntElt`

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): RngOrd, RngOrdIdl -> RngIntElt`

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): RngOrdIdl -> RngIntElt`

`Norm(I): RngOrdFracIdl -> FldRatElt`

`Norm(I): RngOrdFracIdl -> RngOrdFracIdl`

`Norm(I): RngIntFracIdl -> RngIntElt`

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): RngOrdIdl -> RngElt`

Given an ideal I of an order in some number field, the function returns the least positive integer contained in the ideal.

`Minimum(I): RngOrdFracIdl -> RngElt`

`Min(I): RngOrdFracIdl -> RngElt`

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): RngOrdIdl -> RngIntElt`

`AbsoluteNorm(I): RngOrdFracIdl -> FldRatElt`

`NormAbs(I): RngOrdIdl -> RngIntElt`

`NormAbs(I): RngOrdFracIdl -> FldRatElt`

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): RngOrdIdl -> RngIntElt`

`CoefficientHeight(I): RngOrdFracIdl -> RngIntElt`

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): RngOrdIdl -> RngIntElt`

CoefficientLength(I): RngOrdFracIdl -> RngIntElt

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): RngOrdIdl, RngIntElt -> RngIntElt

RamificationDegree(I, p): RngOrdIdl, RngIntElt -> RngIntElt

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): RngOrdIdl -> RngIntElt

RamificationIndex(I): RngOrdIdl -> RngIntElt

Computes the relative ramification index of the prime ideal I over the coefficient ring. To be more precise: Let I be a prime ideal of some order O with coefficient 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.

AbsoluteRamificationDegree(I): RngOrdIdl -> RngIntElt

AbsoluteRamificationIndex(I): RndOrdIdl -> RngIntElt

Return the ramification index of the prime ideal I as an extension of the prime integer which is its minimum.

ResidueClassField(O, I): RngOrd, RngOrdIdl -> FldFin, Map

ResidueClassField(I): RngOrdIdl -> FldFin, Map

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): RngOrdIdl -> RngIntElt

InertiaDegree(I): RngOrdIdl -> RngIntElt

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.

AbsoluteInertiaDegree(I): RngOrdIdl -> RngIntElt

AbsoluteInertiaIndex(I): RngOrdIdl -> RngIntElt

Return the inertia index of the prime ideal I as an extension of the prime integer which is its minimum.

Valuation(I, p): RngOrdFracIdl, RngOrdIdl -> RngIntElt

Valuation(I, p): RngIntFracIdl, RngInt -> RngIntElt

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. If p has been constructed using the Montes algorithm then this valuation computation will also use the Montes algorithm [Sta18].

Content(I): RngOrdFracIdl -> RngIntElt

The content of the ideal I, i.e. the maximal ideal of the base ring dividing I.

Example: Invariants of an Ideal (ex-f01e3a)

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
```

(continues on next page)

(continued from previous page)

```
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
> Norm(Norm(I));
1554547407859179345817 6/317707036579795566191 4307
> FactorizationOfQuotient($1);
[ <2, 10>, <19, 15>, <43, -15> ]
> Norm(Norm(O.3));
1024
> Norm(M.4);
1
> Denominator(I);
43
> Denominator(O.3);
1
> PrimitiveElement(I);
19/43*M.4*O.3
> Minimum(I);
19/43
> [ Norm(Norm(tuple[1])) : tuple in Factorization(I) ];
[ 2, 4, 4, 43, 43, 43, 6859, 3418801, 3418801, 3418801, 2213314919066161 ]
```

39.11.3 Basis Representation

The basis of an ideal can be computed as well as related matrices.

Basis(I): RngOrdIdl -> [RngOrdElt]

Basis(I): RngOrdFracIdl -> [FldOrdElt]

Basis(I): RngIntFracIdl -> [FldRatElt]

Basis(I, R): RngOrdFracIdl, Rng -> [RngElt]

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): RngOrdFracIdl -> MtrxSpcElt

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 $\mathbb{Z}$ for an integral ideal of an order over $\mathbb{Z}$ only or the field of fractions of the coefficient ring of O.

TransformationMatrix(I): RngOrdFracIdl -> MtrxSpcElt, RngIntElt

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 $\mathbb{Z}$ for an integral ideal of an order over $\mathbb{Z}$ or the field of fractions of the coefficient ring of O.

CoefficientIdeals(I): RngOrdFracIdl -> [RngOrdFracIdl]

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: Basis of an Ideal (ex-49f139)

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): RngOrdFracIdl -> ModDed, Map`

For an ideal I in some relative extension, return a Dedekind module over the coefficient ring with the "same" basis.

39.11.4 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): RngOrdIdl -> [ RngOrdElt ]`

`Generators(I): RngOrdFracIdl -> [ FldOrdElt ]`

`Generators(I): RngIntFracIdl -> [ FldRatElt ]`

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): RngOrdFracIdl -> FldOrdElt, FldOrdElt`

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): RngOrdIdl -> RngOrdElt, RngOrdElt, RngIntElt`

Given an integral ideal I of O (a maximal order over $\mathbb{Z}$), 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: Two-Element Presentation of an Ideal (ex-4c86cb)

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
```

39.11.5 Standard Names

These utility functions implement a standard naming convention, which is used in the LMFDB (L-function database) and elsewhere. *Important:* the label of an ideal *depends* on the defining polynomial of the field (but not on any other choices).

`LMFDBLabel(I): RngOrdIdl -> MonStgElg`

The name of the ideal I of a maximal order. This is a well-defined function of the ideal and the defining polynomial of the number field (which must be a simple extension of $\mathbb{Q}$).

`LMFDBIdeal(K, s): FldNum, MonStgElt -> RngOrdIdl`

The ideal of $\mathbb{Z}_K$ that has the name s.

39.11.6 Predicates on Ideals

Ideals may be tested for various properties and whether given elements lie in the ideal.

`I eq J: RngOrdFracIdl, RngOrdFracIdl -> BoolElt`

`I ne J: RngOrdFracIdl, RngOrdFracIdl -> BoolElt`

`x in I: FldOrdElt, RngOrdFracIdl -> BoolElt`

`x in I: RngOrdElt, RngOrdFracIdl -> BoolElt`

`x notin I: FldOrdElt, RngOrdFracIdl -> BoolElt`

`x notin I: RngOrdElt, RngOrdFracIdl -> BoolElt`

`I subset J: RngOrdIdl, RngOrdIdl -> BoolElt`

`IsIntegral(I): RngOrdFracIdl -> BoolElt`

`IsIntegral(I): RngIntFracIdl -> BoolElt`

Returns `true` if and only if the fractional ideal I is integral.

`IsZero(I): RngOrdFracIdl -> BoolElt`

Returns `true` if the ideal I is the zero ideal, `false` otherwise.

`IsOne(I): RngOrdIdl -> BoolElt`

`IsOne(I): RngOrdFracIdl -> BoolElt`

`IsOne(I): RngIntFracIdl -> BoolElt`

Returns `true` if the ideal I is the ideal generated by 1, `false` otherwise.

`IsPrime(I): RngOrdIdl -> BoolElt, RngOrdIdl`

`IsPrime(I): RngOrdFracIdl -> BoolElt`

Returns `true` if and only if the ideal I is a prime ideal of `Order(I)`. If `Order(I)` is maximal and I is a proper ideal of type `RngOrdIdl`, then in the case I is not prime, the function also returns a proper divisor.

`IsPrincipal(I): RngOrdFracIdl -> BoolElt, FldOrdElt`

```
ClassGroup: Verbose                    Default: 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. If I is an ideal of a non-maximal order the Unit group may be required also. Class group and unit group computations will be carried out to the most rigorous proof level, if this level of proof is not required then the *UnitGroup* should be precomputed and *ClassGroup* bounds set before calling `IsPrincipal`.

`IsRamified(P): RngOrdIdl -> BoolElt`

Returns `true` iff the ramification index of the prime ideal P is greater than 1.

`IsRamified(P, O): RngOrdIdl, RngOrd -> BoolElt`

`IsRamified(P, O): RngIntElt, RngOrd -> BoolElt`

`IsRamified(P, O): RngInt, RngOrd -> BoolElt`

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).

`IsSquarefree(I): RngOrdIdl -> BoolElt`

`IsSquarefree(I): RngInt -> BoolElt`

Returns `true` iff the ideal I has no repeated factors.

`IsTotallyRamified(P): RngOrdIdl -> BoolElt`

Returns `true` iff the ramification index of the prime ideal P equals the degree of its order over the base field.

`IsTotallyRamified(P, O): RngOrdIdl, RngOrd -> BoolElt`

`IsTotallyRamified(P, O): RngIntElt, RngOrd -> BoolElt`

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): FldAlg -> BoolElt`

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): RngOrd -> BoolElt`

Returns `true` if all primes dividing the discriminant of the maximal order O are totally ramified over the coefficient ring of O.

`IsWildlyRamified(P): RngOrdIdl -> BoolElt`

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): RngOrdIdl, RngOrd -> BoolElt`

`IsWildlyRamified(P, O): RngIntElt, RngOrd -> BoolElt`

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): RngOrdIdl -> BoolElt`

Returns `true` iff the prime ideal P is not wildly ramified.

`IsTamelyRamified(P, O): RngOrdIdl, RngOrd -> BoolElt`

`IsTamelyRamified(P, O): RngIntElt, RngOrd -> BoolElt`

Returns `true` iff the prime ideal or integer P is not wildly ramified in the order O.

`IsUnramified(P): RngOrdIdl -> BoolElt`

Returns `true` iff the ramification index of the prime ideal P is 1.

`IsUnramified(P, O): RngOrdIdl, RngOrd -> BoolElt`

`IsUnramified(P, O): RngIntElt, RngOrd -> BoolElt`

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): RngOrdIdl -> BoolElt`

Returns `true` iff the inertia degree of the prime ideal P is the field degree.

`IsInert(P, O): RngOrdIdl, RngOrd -> BoolElt`

`IsInert(P, O): RngIntElt, RngOrd -> BoolElt`

`IsInert(P, O): RngInt, RngOrd -> BoolElt`

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): RngOrdIdl -> BoolElt`

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): RngOrdIdl, RngOrd -> BoolElt`

`IsSplit(P, O): RngIntElt, RngOrd -> BoolElt`

`IsSplit(P, O): RngInt, RngOrd -> BoolElt`

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): RngOrdIdl -> BoolElt`

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): RngOrdIdl, RngOrd -> BoolElt`

`IsTotallySplit(P, O): RngIntElt, RngOrd -> BoolElt`

`IsTotallySplit(P, O): RngInt, RngOrd -> BoolElt`

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.

39.11.7 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: RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

`I * J: RngIntFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

`I * J: RngIntFracIdl, RngIntFracIdl -> RngIntFracIdl`

The product IJ of the (fractional) ideals I and J, generated by the products of elements in I and elements in J.

`x * I: RngElt, RngOrdFracIdl -> RngOrdFracIdl`

`x * I: FldRatElt, RngIntFracIdl -> RngIntFracIdl`

`x * I: RngIntElt, RngIntFracIdl -> RngIntFracIdl`

`I * x: RngOrdFracIdl, RngElt -> RngOrdFracIdl`

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: [RngOrdFracIdl] -> RngOrdFracIdl`

`&* L: { RngOrdFracIdl } -> RngOrdFracIdl`

The product of all ideals in the sequence or set L.

`I div J: RngOrdIdl, RngOrdIdl -> RngOrdIdl`

`I div J: RngOrdFracIdl, RngOrdFracIdl -> RngOrdIdl`

`I div J: RngInt, RngInt -> RngInt`

For ideals I and J of some maximal order such that J divides I (or equivalently that J is contained in I), return the integral ideal K such that $JK = I$.

`I / J: RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

The quotient of the (fractional) ideals I and J of a maximal order O. This is the fractional ideal K of O with the property that $JK = I$.

There are some considerations to be made here. Firstly, the "I/J" notation is interpreted as above, and not (say) as an ideal of the quotient O/J (see also the example with $\mathbf{Z}$ as a number field order in Section Ideals of {\mathbb{Z}} and {\mathbb{Z}} as a Number Field Order). Secondly, since I/J is only defined (in Magma) for *maximal* orders, this is the same (in Magma) as the `ColonIdeal` (or `IdealQuotient`) as noted below. Thirdly, I/J and `I div J` are not synonyms, as I/J will work even when J does not divide I, while `I div J` will give an error in that case.

`I / x: RngOrdFracIdl, RngElt -> RngOrdFracIdl`

Given an ideal I and an element x construct the fractional ideal I/x.

`I + J: RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

`I + J: RngIntFracIdl, RngIntFracIdl -> RngIntFracIdl`

`I + J: RngIntFracIdl, RngInt -> RngIntFracIdl`

The sum of the (fractional) ideals I and J, generated by the sums of elements in I and elements in J.

`I ^ k: RngOrdFracIdl, RngIntElt -> RngOrdFracIdl`

`I ^ k: RngIntFracIdl, RngIntElt -> RngIntFracIdl`

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: RngOrdFracIdl, RngOrdFracIdl -> BoolElt`

`I eq J: RngIntFracIdl, RngIntFracIdl -> BoolElt`

`I eq J: RngIntFracIdl, RngInt -> BoolElt`

Tests if the ideals I and J are equal.

`I subset J: RngOrdIdl, RngOrdIdl -> BoolElt`

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: RngOrdElt, RngOrdIdl -> BoolElt`

`E in I: RngOrdElt, RngOrdFracIdl -> BoolElt`

`E in I: FldAlgElt, RngOrdFracIdl -> BoolElt`

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): RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

`Lcm(I, J): RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

`LeastCommonMultiple(I, J): RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

Return the least common multiple of ideals I and J. They must both be of the same maximal order.

`GCD(I, J): RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

`Gcd(I, J): RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

`GreatestCommonDivisor(I, J): RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

The greatest common divisor of the ideals I and J of some maximal order of an algebraic number field.

`Content(M): Mtrx -> RngOrdFracIdl`

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: RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

The intersection of the (fractional) ideals I and J. For ideal in the maximal order this is the same as the lcm.

`&meet S: [RngOrdFracIdl] -> RngOrdFracIdl`

`&meet S: { RngOrdFracIdl } -> RngOrdFracIdl`

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: RngOrdFracIdl, Rng -> Any`

`R meet I: Rng, RngOrdFracIdl -> Any`

The intersection of the ideal I with the compatible ring R. If $R = \mathbb{Q}$ an error will occur since ideals of $\mathbb{Q}$ cannot be created. If such information is required use `Minimum` instead. An ideal of R is returned.

`a mod I: RngOrdElt, RngOrdIdl -> RngOrdElt`

A representative of the element a of an order O in the quotient O/I.

`InverseMod(E, M): RngOrdElt, RngIntElt -> RngOrdElt`

`InverseMod(E, M): RngOrdElt, RngOrdIdl -> RngOrdElt`

`Modinv(E, M): RngOrdElt, RngOrdIdl -> RngOrdElt`

`Modinv(E, M): RngOrdElt, RngIntElt -> RngOrdElt`

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): RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

`IdealQuotient(I, J): RngOrdFracIdl, RngOrdFracIdl -> RngOrdFracIdl`

In Magma, the colon ideal (or ideal quotient) $[I : J]$ for two fractional ideals I, J is defined as the fractional ideal $(I : J) = \{x \in \mathrm{Frac}(O) : xJ \subseteq I\}$, where O is the order to which I and J belong. In this definition we have $x \in \mathrm{Frac}(O)$ rather than $x \in O$, and so this differs from `ColonIdeal` for other structures, where the result would be an ideal rather than a fractional ideal.

For ideals of a maximal order (or in general for *invertible* ideals) the `ColonIdeal` is equivalent to I/J (see above), otherwise only $J(I : J) \subset I$ holds. Below is an example of the latter.

Currently `ColonIdeal` and `IdealQuotient` do not exist for `RngInt` types, so there is a slight incompatibility for **Z** as a number field order (see §{\mathbb{Z}} as a Number Field Order).

Example: Colon Ideal of a Noninvertible Ideal `(ex-9af5d2)`

```
> O := EquationOrder(Polynomial([4,0,1]));
> z := O.2; // z^2 = -4
> I := ideal<O|1>; // O as an ideal
> J := ideal<O|[2,z]>; // noninvertible ideal
> ColonIdeal(I,J)*J eq I;
false
```

`IntegralSplit(I): RngOrdFracIdl -> RngOrdIdl, RngElt`

Given an ideal I, return an integral ideal J and a minimal positive integer d such that $I = J/d$.

`Different(I): RngOrdFracIdl -> RngOrdFracIdl`

The different of the (possibly fractional) ideal I of an order of an algebraic number field.

`Codifferent(I): RngOrdFracIdl -> RngOrdFracIdl`

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.

39.11.8 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): RngOrdFracIdl, RngIntElt -> RngOrdFracIdl`

Find the kth root of the ideal I if it exists.

`IsPower(I, k): RngOrdFracIdl, RngIntElt -> BoolElt, RngOrdFracIdl`

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): RngOrdFracIdl -> RngOrdFracIdl`

`Sqrt(I): RngOrdFracIdl -> RngOrdFracIdl`

Return the square root of the ideal I if I is square.

`IsSquare(I): RngOrdFracIdl -> BoolElt, RngOrdFracIdl`

Return `true` if the ideal I is the square of an ideal and the ideal it is a square of otherwise `false`.

39.11.9 Factorization and Primes

The factorization of an ideal into prime ideals and the divisors of an ideal can be determined.

`Decomposition(O, p): RngOrd, RngIntElt -> [<RngOrdIdl, RngIntElt>]`

```
Al: MonStgElt                    Default:
```

`Decomposition(O, p): RngOrd, RngOrdIdl -> [ <RngOrdIdl, RngIntElt> ]`

```
IdealDecompose: Verbose                    Default: 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. The parameter `Al` can be set to `"Montes"` when O is an extension of $\mathbb{Z}$ by a single monic integral polynomial and p is an integer, in which case the Montes algorithm [Sta18] will be used to compute the decomposition.

`DecompositionType(O, p): RngOrd, RngIntElt -> [<RngIntElt, RngIntElt>]`

`DecompositionType(O, p): FldAlg, RngIntElt -> [<RngIntElt, RngIntElt>]`

`DecompositionType(O, p): FldAlg, RngOrdIdl -> [<RngIntElt, RngIntElt>]`

`DecompositionType(O, p): RngOrd, RngOrdIdl -> [ <RngIntElt, RngIntElt> ]`

```
IdealDecompose: Verbose                    Default: 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): RngOrdFracIdl -> [<RngOrdIdl, RngIntElt>]`

`Factorisation(I): RngOrdFracIdl -> [<RngOrdIdl, RngIntElt>]`

`Factorization(I): RngIntFracIdl -> [<RngInt, RngIntElt>]`

`Factorisation(I): RngIntFracIdl -> [<RngInt, RngIntElt>]`

```
IdealDecompose: Verbose                    Default: 5
```

Returns the prime ideal factorization of an ideal I in a maximal order O, as a sequence of 2-tuples (prime ideal and integer exponent).

Example: Factorization in a Non-Maximal Order `(ex-4f7652)`

We provide an example of `Factorization`. An ideal in an order which is not maximal can only be decomposed into primary ideals. We illustrate how to compute some prime "factors" of an ideal of an order which is not maximal.

```
> R<x>:=PolynomialRing(Integers());
> f:=x^3+4*x+15;
> K<y>:=NumberField(f);
> E:=EquationOrder(K);
> O:=MaximalOrder(K);
> c:=4*y^2+6*y+4;
> I:=c*O;
> Factorisation(I);
[
    <Prime Ideal of E
    Two element generators:
        [2, 0, 0]
        [1, 1, 0], 1>,
    <Prime Ideal of E
    Two element generators:
        [2, 0, 0]
        [3, 1, 1], 1>,
    <Prime Ideal of E
    Two element generators:
        [3, 0, 0]
        [1, 0, 1], 1>,
    <Prime Ideal of E
    Two element generators:
        [151, 0, 0]
        [262, 1, 0], 1>
]
> J := c*E;
> Factorisation(J);
>>     Factorisation(J);
                      ^
Runtime error in 'Factorisation': Order associated with ideal argument 1 is not
known to be maximal
> E2 := Decomposition(E, 2);
> E2;
[
    <Prime Ideal of E
    Two element generators:
        [2, 0, 0]
        [1, 1, 0], 1>,
    <Prime Ideal of E
    Two element generators:
        [2, 0, 0]
        [3, 1, 1], 1>
]
> Valuation(J, E2[1][1]);
1
> Valuation(J, E2[2][1]);
1
> J := ColonIdeal(J, E2[1][1]); J;
Ideal of E
Basis:
[  1   3 403]
[  0   6 702]
[  0   0 906]
> J := ColonIdeal(E!!J, E2[2][1]); J;
Ideal of E
Basis:
```

(continues on next page)

(continued from previous page)

```
[  1   0  52]
[  0   3 351]
[  0   0 453]
> E3 := Decomposition(E, 3);
> E3;
[
    <Prime Ideal of E
    Two element generators:
        [3, 0, 0]
        [0, 1, 0], 1>,
    <Prime Ideal of E
    Two element generators:
        [3, 0, 0]
        [1, 0, 1], 1>
]
> Valuation(J, E3[1][1]);
0
> Valuation(J, E3[2][1]);
1
> J := ColonIdeal(E!!J, E3[2][1]); J;
Ideal of E
Basis:
[  1   0  52]
[  0   1 117]
[  0   0 151]
> E151 := Decomposition(E, 151);
> E151;
[
    <Prime Ideal of E
    Two element generators:
        [151, 0, 0]
        [18, 1, 0], 1>,
    <Prime Ideal of E
    Two element generators:
        [151, 0, 0]
        [22, 1, 0], 1>,
    <Prime Ideal of E
    Two element generators:
        [151, 0, 0]
        [262, 1, 0], 1>
]
> Valuation(J, E151[1][1]);
0
> Valuation(J, E151[2][1]);
0
> Valuation(J, E151[3][1]);
1
> J := ColonIdeal(E!!J, E151[3][1]); J;
Ideal of E
Basis:
[1 0 0]
[0 1 0]
[0 0 1]
```

It will not always be the case that an ideal of a non maximal order will be able to be factorized into powers of prime ideals, although in this example it is the case.

Divisors(I): RngOrdIdl -> [<RngOrdIdl, RngIntElt>]

Return the ideals which divide the ideal I which must be of a maximal order.

`Support(I): RngOrdFracIdl -> {RngOrdIdl}`

`Support(I): RngIntFracIdl -> {RngInt}`

For a non-zero ideal I of some maximal order, return the set of prime ideals p dividing I.

`Support(L): [RngOrdFracIdl] -> {RngOrdIdl}`

`Support(L): [FldAlgElt] -> {RngOrdIdl}`

```
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): [RngOrdFracIdl] -> RngOrdIdl`

```
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): [RngOrdIdl], RngOrdFracIdl`

```
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): [RngOrdFracIdl], [RngIntElt] -> RngOrdFracIdl`

Given sequences B of ideals of some maximal order and E of integers, compute the ideal $\prod B[i]^{E[i]}$.

39.11.10 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* and Chapter p-Adic Rings and Their Extensions).

`ChineseRemainderTheorem(I1, I2, e1, e2): RngOrdIdl, RngOrdIdl, RngOrdElt, RngOrdElt -> RngOrdElt`

`ChineseRemainderTheorem(X, M): SeqEnum[RngOrdElt], SeqEnum[RngOrdIdl] -> RngOrdElt`

`CRT(I1, I2, e1, e2): RngOrdIdl, RngOrdIdl, RngOrdElt, RngOrdElt -> RngOrdElt`

`CRT(X, M): SeqEnum[RngOrdElt], SeqEnum[RngOrdIdl] -> RngOrdElt`

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): RngOrdIdl, [RngIntElt], RngOrdElt, [RngIntElt] -> RngOrdElt`

`ChineseRemainderTheorem(I1, L1, e1, L2): RngOrdIdl, [RngIntElt], RngOrdElt, [RngIntElt] -> RngOrdElt`

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 $L2[i]$.

`WeakApproximation(I, V): [RngOrdIdl], [RngIntElt] -> FldOrdElt`

`WeakApproximation(I, V): [RngInt], [RngIntElt] -> FldRatElt`

Compute an element in the field of fractions of the order of the ideals in I which has valuation $V[i]$ at the prime ideal $I[i]$.

`Idempotents(I, J): RngOrdIdl, RngOrdIdl -> BoolElt, RngOrdElt, RngOrdElt`

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): RngOrdIdl, RngOrdIdl -> FldOrdElt`

`MakeCoprime(I, J): RngOrdIdl, RngOrdIdl -> FldOrdElt`

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.

`IdealsUpTo(B, O): RngIntElt, RngOrd -> [RngOrdIdl]`

`IdealsUpTo(B, K): RngIntElt, FldNum -> [RngOrdIdl]`

`CoprimeTo: RngOrdIdl`	`Default:`

Given a number field K or a maximal order O, returns a sequence containing the ideals of O or the maximal order of K with norm at most B which are coprime to any ideals given in `CoprimeTo`.

`ClassRepresentative(I): RngOrdFracIdl -> RngOrdFracIdl`

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.

`SUnitGroup(I): RngOrdFracIdl -> GrpAb, Map`

`SUnitGroup(S): [ RngOrdIdl ] -> GrpAb, Map`

`Raw : BoolElt`		`Default: false`
`ClassGroup: Verbose`		`Default: 5`

This function computes the group of S-units for the set S of prime ideals, where S may be input as a sequence or as an ideal (in which case S is the set of primes appearing in its factorization). An element mu is an S-unit iff $v_p(mu) = 0$ for all primes p not in S.

The function returns (by default) an abstract abelian group A, and a map from A to the field.

When `Raw` is `true`, the S-units are represented as power-products rather than as ordinary elements. A fixed sequence L of field elements is given, and each unit is specified as a vector of integers e (of the same length as L) such that the unit equals $\prod L_i^{e_i}$. Three values are returned in the `Raw` case: an abstract abelian group A, a map from A to exponent vectors, and the sequence L.

Example: S-Units of a Real Quadratic Field (ex-062dcc)

First we compute the 3-units of $\mathbb{Q}(\sqrt{10})$.

```
> 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 : Proof:="GRH");
> U, mU, base := SUnitGroup(1*L:Raw);
> logs := Matrix([Logs(x) : x in Eltseq(base)]);
> mU(U.3)*logs;
(2815256.998090806477937318458440358713392011019115562
    -636746.052519109818325583483504897441603096166734 57
    -770882.446526293420643075745715281915092909342801 66)
```

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): Map, Map, SeqEnum[RngOrdIdl] -> Map`

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): Map, SeqEnum[Map], SeqEnum[RngOrdIdl] -> [Map]`

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): Map, FldAlgElt, SeqEnum[RngOrdIdl] -> GrpAbElt`

`SUnitDiscLog(SU, L, S): Map, SeqEnum[FldAlgElt], SeqEnum[RngOrdIdl] -> GrpAbElt`

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: Galois Action on S-Units `(ex-95d956)`

```
> 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<FieldOfFractions(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]
```

39.11.11 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.

Operations on Quotient Rings

The creation of quotient rings and the functions which may be applied to them are described.

`quo< O | I >: RngOrd, RngOrdIdl -> RngOrdRes`

`quo< O | M >: RngOrd, ModDed -> RngOrdRes`

`quo< O | M >: RngOrd, AlgMatElt -> RngOrdRes`

`quo< O | S >: RngOrd, RngElt, ..., RngElt -> RngOrdRes`

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.

`quo< O | m >: RngOrd, RngIntElt -> RngOrdRes`

Construct the quotient ring $Q = O/m$ where m is a prime power.

`UnitGroup(OQ): RngOrdRes -> GrpAb, Map`

`MultiplicativeGroup(OQ): RngOrdRes -> GrpAb, Map`

Returns an abelian group and the map from the group into OQ, a quotient of an absolute maximal order.

`Modulus(OQ): RngOrdRes -> RngOrdIdl`

Return the denominator of the quotient ring OQ, i.e. I where $OQ = O/I$.

Example: Quotient Rings of an Order (ex-d59aa1)

Creation of quotient rings is 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]]
```

Elements of Quotients

Functions for elements of the quotient rings are predominantly arithmetic. Elements of quotient rings can be looped through using `for x in OQ do ... end for`.

`OQ ! a: RngOrdRes, Elt -> RngOrdResElt`

Coerce the element a into the quotient OQ where a is anything that can be coerced into the order OQ is a quotient of.

`Random(OQ): RngOrdRes -> RngOrdResElt`

A random element of the quotient ring OQ.

`a mod I: RngOrdElt, RngOrdIdl -> RngOrdElt`

A canonical representative of the element a (belonging to an order O) in the quotient ring O/I.

`a * b: RngOrdResElt, RngOrdResElt -> RngOrdResElt`

`a + b: RngOrdResElt, RngOrdResElt -> RngOrdResElt`

`a - b: RngOrdResElt, RngOrdResElt -> RngOrdResElt`

`a / b: RngOrdResElt, RngOrdResElt -> RngOrdResElt`

```
- a: RngOrdResElt -> RngOrdResElt
a ^ n: RngOrdResElt, RngIntElt -> RngOrdResElt
a eq b: RngOrdResElt, RngOrdResElt -> BoolElt
a ne b: RngOrdResElt, RngOrdResElt -> BoolElt
a div b: RngOrdResElt, RngOrdResElt -> RngOrdResElt
a mod b: RngOrdResElt, RngOrdResElt -> RngOrdResElt
Quotrem(a, b): RngOrdResElt, RngOrdResElt -> RngOrdResElt, RngOrdResElt
Lcm(a, b): RngOrdResElt, RngOrdResElt -> RngOrdResElt
Gcd(a, b): RngOrdResElt, RngOrdResElt -> RngOrdResElt
XGcd(a, b): RngOrdResElt, RngOrdResElt -> RngOrdResElt, RngOrdResElt, RngOrdResElt
IsZero(a): RngOrdResElt -> BoolElt
```

Returns `true` if and only if the quotient ring element a is the zero element of the quotient ring OQ.

```
IsOne(a): RngOrdResElt -> BoolElt
```

Returns `true` if and only if the quotient ring element a is the one element of the quotient ring OQ.

```
IsMinusOne(a): RngOrdResElt -> BoolElt
```

Returns `true` if and only if the quotient ring element a is the minus one element of the quotient ring OQ.

```
IsUnit(a): RngOrdResElt -> BoolElt
```

Returns `true` if and only if the quotient ring element a has an inverse in the quotient ring OQ.

```
Eltseq(a): RngOrdResElt -> []
ElementToSequence(a): RngOrdResElt -> []
```

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.

```
EuclideanNorm(a): RngOrdResElt -> RngIntElt
```

Return the Euclidean norm of the element a of a quotient ring OQ, where the Euclidean norm is the function that makes OQ into a Euclidean ring.

```
Annihilator(a): RngOrdResElt -> RngOrdResElt
```

The annihilator of the element a in the quotient ring OQ.

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 [FF].

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): RngOrdIdl, RngIntElt -> RngOrdRecoEnv
ReconstructionEnvironment(p, k): RngIntElt, RngIntElt -> RngOrdRecoEnv
```

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): RngOrdElt, RngOrdRecoEnv -> RngOrdElt
Reconstruct(x, R): RngIntElt, RngOrdRecoEnv -> RngIntElt
```

```
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): RngOrdRecoEnv, RngIntElt`

Change the ideal $I = p^l$ stored in R to p^k.

Example: Roots via Order Reconstruction (ex-c0c980)

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
 ]
```

39.12 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 $\mathbb{R}$ or with pairs of embeddings into $\mathbb{C}$.

The group of divisors is formally the free group generated by the finite places and the $\mathbb{R}$-vectorspace generated by the infinite ones.

For more information see Section *Places and Divisors*.

39.12.1 Creation of Structures

`Places(K): FldNum -> PlcNum`

`DivisorGroup(K): FldNum -> DivNum`

The set of places of the number field K and the group of divisors of K respectively.

39.12.2 Operations on Structures

`d1 eq d2: DivNum, DivNum -> BoolElt`

`p1 eq p2: PlcNum, PlcNum -> BoolElt`

`NumberField(P): PlcNum -> FldNum`

`NumberField(D): DivNum -> FldNum`

The number field for which P is the set of places or D is the group of divisors.

39.12.3 Creation of Elements

`Place(I): RngOrdIdl -> PlcNumElt`

The place corresponding to prime ideal I.

`Decomposition(K, I): FldAlg, Infty -> SeqEnum`

`Decomposition(K, p): FldAlg, RngIntElt -> SeqEnum`

Al: MonStgElt	Default:

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. When K is an extension of $\mathbb{Q}$ by a single monic integral polynomial and p is an integer, `Al` can be set to `"Montes"` to use the Montes algorithm [Sta18] for this computation.

`Decomposition(K, p): FldNum, PlcNumElt -> SeqEnum`

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): Map[FldRat, FldAlg], RngIntElt -> SeqEnum[<PlcNumElt, RngIntElt>]`

`Decomposition(m, p): Map[FldAlg, FldAlg], PlcNumElt -> SeqEnum[<PlcNumElt, RngIntElt>]`

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): FldAlg -> [PlcNumElt]`

`InfinitePlaces(O): RngOrd -> [PlcNumElt]`

A sequence containing all the infinite places of the number field K or the order O is returned.

`RealPlaces(K): FldAlg -> [PlcNumElt]`

The sequence of infinite places of K which correspond to real embeddings.

`Divisor(pl): PlcNumElt -> DivNumElt`

The divisor $1 * pl$ for a place pl.

`Divisor(I): RngOrdFracIdl -> DivNumElt`

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): FldNumElt -> DivNumElt`

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.

39.12.4 Arithmetic with Places and Divisors

Divisors and places can be added, negated, subtracted and multiplied and divided by integers.

`d1 + d2: DivNumElt, DivNumElt -> DivNumElt`

`p + d: PlcNumElt, DivNumElt -> DivNumElt`

`d + p: DivNumElt, PlcNumElt -> DivNumElt`

`p1 + p2: PlcNumElt, PlcNumElt -> DivNumElt`

`- p: PlcNumElt -> DivNumElt`

`- d: DivNumElt -> DivNumElt`

`d - p: DivNumElt, PlcNumElt -> DivNumElt`

`p - d: PlcNumElt, DivNumElt -> DivNumElt`

`d1 - d2: DivNumElt, DivNumElt -> DivNumElt`

`p1 - p2: PlcNumElt, PlcNumElt -> DivNumElt`

`p * k: PlcNumElt, RngIntElt -> DivNumElt`

`d * k: DivNumElt, RngIntElt -> DivNumElt`

`p div k: PlcNumElt, RngIntElt -> DivNumElt`

`d div k: DivNumElt, RngIntElt -> DivNumElt`

39.12.5 Other Functions for Places and Divisors

`Valuation(a, p): FldNumElt, PlcNumElt -> RngElt`

`Valuation(a, p): RngElt, PlcNumElt -> RngElt`

The valuation of the element a of a number field or order at the place p. If p was constructed using the Montes algorithm then the Montes algorithm [Sta18] will also be used to compute the valuation.

`Valuation(I, p): RngOrdFracIdl, PlcNumElt -> RngElt`

The valuation of the ideal I at the finite place p.

`Support(D): DivNumElt -> SeqEnum, SeqEnum`

The support of the divisor D as a sequence of places and a sequence of the corresponding exponents.

`Ideal(D): DivNumElt -> RngOrdIdl`

`Ideal(D): PlcNumElt -> RngOrdIdl`

The ideal corresponding to the finite part of the divisor D.

`IsFinite(p): PlcNumElt -> BoolElt`

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): PlcNumElt -> BoolElt, RngIntElt`

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.

`IsReal(p): PlcNumElt -> BoolElt`

For an infinite place, returns `true` if the image of the embedding is contained in the real numbers.

`IsComplex(p): PlcNumElt -> BoolElt`

For an infinite place, return `true` if the image of the embedding is not contained in the real numbers.

`Extends(P, p): PlcNumElt, PlcNumElt -> BoolElt`

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 divides, 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): PlcNumElt -> RngIntElt`

`Degree(P): PlcNumElt -> RngIntElt`

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): DivNumElt -> RngElt`

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): PlcNumElt -> FldNum`

`NumberField(D): DivNumElt -> FldNum`

For a place P or divisor D of a number field, return the underlying number field.

`ResidueClassField(P): PlcNumElt -> Fld`

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): PlcNumElt -> FldNumElt`

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): PlcNumElt -> RngIntElt`

The degree of the completion at the place P, ie. the product of the inertia degree times the ramification index.

`RamificationIndex(P): PlcNumElt -> RngIntElt`

The ramification index of the place P. For infinite real places this is 1 and 2 for complex places.

`DecompositionGroup(P): PlcNumElt -> GrpPerm`

For a place P of a normal number field, return the decomposition group as a subgroup of the (abstract) automorphism group.

39.13 The Montes Algorithm

The Montes algorithm can be considered as a factorization algorithm for polynomials over local fields. For a defining polynomial f of a number field K and a prime number p the Montes algorithm produces OM representations for all prime ideals of K lying of over p, which parametrize the irreducible factors of f over the p-adic numbers. The OM representation for prime ideals leads to an OM presentation of fractional ideals. The basic algorithms are explained in [GMN13] These OM representations allow the construction of ideals and divisors without the computation of a maximal order, which the construction of ideals of types `RngOrdIdl` and `RngOrdFracIdl` require. The Montes algorithm can also be used to efficiently compute integral bases which can be use to construct maximal orders.

This section explains the use of the Montes Algorithm. For how to use the Montes algorithm to compute maximal orders of number fields see Section *Maximal Orders*, maximal orders of function fields see Section *Construction of Orders of Algebraic Function Fields*, decomposition of primes in number fields see Sections *Factorization and Primes* and *Creation of Elements* and decomposition of primes in function fields see Sections *Further Ideal Operations* and *Creation of Elements*.

Montes(f, p): RngUPolElt, RngElt -> SeqEnum, SeqEnum, RngIntElt

Field: BoolElt	Default: false

Let f be a monic polynomial with integer or polynomial coefficients and p be a prime number. The Montes algorithm outputs the triple:

1. A sequence of OM representations for the p-adic irreducible factors of f.
2. A sequence of intervals with the position of the prime ideals in each disconnected tree of OM representations.
3. The p-index of f.

If the parameter **Field** is set to **true** the ψ polynomial at the last level of each type of each OM representation will be computed.

Example: Montes Algorithm on a Polynomial (ex-25d08e)

Here's an example of how to use it:

```
> _<x> := PolynomialRing(Integers());
> f := x^5 + 343*x^4 + 49*x^3 + 343*x^2 + 7*x + 6;
> p := 7;
> Montes(f,p);
[

    ,

]
[
    [ 1 ],
    [ 2 ]
]
0
```

The Montes algorithm can also be called for a number field K if it is defined by a monic polynomial with integer coefficients.

Montes(K, p): FldArith, RngElt

The Montes algorithm is applied to the defining polynomial of the number field K at the prime p.

1. A sequence of OM representations for the prime ideals of K is stored in **K`PrimeIdeals[p]**.
2. The p-adic valuation of the index $[\mathbb{Z}_K : \mathbb{Z}[\theta]]$ is stored in **K`LocalIndex[p]**

Example: Montes Algorithm on a Number Field (ex-cb9d42)

Here's an example of how to use it:

```
> _<x> := PolynomialRing(Integers());
> f := x^5 + 343*x^4 + 49*x^3 + 343*x^2 + 7*x + 6;
> K := NumberField(f);
> p := 7;
> Montes(K,p);
> K`PrimeIdeals[p];
[
    OM prime ideal over   7
```

(continues on next page)

(continued from previous page)

```
    of Number Field with defining polynomial x^5 + 343*x^4 + 49*x^3 + 343*x^2 +
    7*x + 6 over the Rational Field
    having residual degree   1
    and ramification index   1
    Last phi polynomial is x + 6,
    OM prime ideal over   7
    of Number Field with defining polynomial x^5 + 343*x^4 + 49*x^3 + 343*x^2 +
    7*x + 6 over the Rational Field
    having residual degree   4
    and ramification index   1
    Last phi polynomial is x^4 + x^3 + x^2 + x + 1
]
> K`LocalIndex[p];
0
```

SFL(P, s): OMIdl, RngIntElt

Perform the single factor lifting algorithm, [GNP12], on the prime ideal P in OM representation until the slope is at least s.

Example: Single Factor Lifting of an OM Prime (ex-e93eb7)

```
> Ax<x> := PolynomialRing(Integers());
> f := x^5 + 343*x^4 + 49*x^3 + 343*x^2 + 7*x + 6;
> L := NumberField(f);
> p := 7;
> Montes(L,p);
> P := L`PrimeIdeals[p,1];
> P`Type[#P`Type]`slope;
1
> P`Type[#P`Type]`Phi; // the approximation
x + 6
> slope := 50;
> SFL(~P,slope:update := true);
> P`Type[#P`Type]`slope;
50
> P`Type[#P`Type]`Phi; // the approximation
x + 33259860663899418450840121798616672813461 8
```

SetUseMontes(f): BoolElt

SetUseMontes(t, f): Cat, BoolElt

Set whether the Montes algorithm is to be used for maximal order and prime decomposition calculations. The type t can be a type which inherits from `FldNum`. This setting can be overwritten by setting the `Al` parameter of the `MaximalOrder` and `Decomposition` intrinsics to `"Montes"`.

GetUseMontes(t): Cat -> BoolElt

Return whether the Montes algorithm will be used by default for computations of maximal orders and decompositions for types inheriting from type t, where t is a type which inherits from `FldNum`.

SetVerbose("Montes", v): MonStgElt, RngIntElt

Set the level for verbose printing for the Montes algorithm to be $v \leq 3$.

39.14 Ideals in OM Representation

Ideals of type `RngOrdFracIdl` can have an OM representation computed from them. Similarly ideals constructed from the Montes algorithm in OM representation can be converted into ideals of type `RngOrdFracIdl`.

`Ideal(I): OMIdl -> RngOrdIdl`

Translates ideal I in OM representation into a Magma representation.

`OMRepresentation(I): RngFunOrdIdl -> OMIdl`

`OMRepresentation(I): RngOrdFracIdl -> OMIdl`

Computes the OM representation of the ideal I in a number field.

`OMRepresentation(L, S): FldArith, [FldArithElt] -> OMIdl`

`OMRepresentation(L, a): FldArith, FldArithElt -> OMIdl`

`OMRepresentation(L, a): FldArith, RngElt -> OMIdl`

Given a number field L and an element a or sequence S of elements in L, construct, in OM representation, the ideal generated by these elements.

Example: Converting to and from OM Representation (ex-738d2c)

```
>       Ax<x> := PolynomialRing(Integers());
>       f := x^5 + 343*x^4 + 49*x^3 + 343*x^2 + 7*x + 6;
>       L := NumberField(f);
>       p := 7;
>       Montes(L,p);
> Ideal(L`PrimeIdeals[p,1]);
Prime Ideal
Two element generators:
    [7, 0, 0, 0, 0]
    [6, 1, 0, 0, 0]
> OMRepresentation(L, [7, 1 + L.1 + L.1^2 + L.1^3 + L.1^4]);
OM ideal of the field Number Field with defining polynomial x^5 + 343*x^4 +
49*x^3 + 343*x^2 + 7*x + 6 over the Rational Field
generated by [
7,
$.1^4 + $.1^3 + $.1^2 + $.1 + 1
]

> Ideal($1);
Ideal
Two element generators:
    [7, 0, 0, 0, 0]
    [1, 1, 1, 1, 1]
```

39.14.1 Ideal Arithmetic

`I + J: OMIdl, OMIdl -> OMIdl`

`I * J: OMIdl, OMIdl -> OMIdl`

`I / J: OMIdl, OMIdl -> OMIdl`

The sum, product or quotient of ideals I and J in OM representation.

`I ^ n: OMIdl, RngIntElt -> OMIdl`

The n-th power of the ideal I in OM representation.

Example: Arithmetic of OM Ideals (ex-58ff3a)

```
> Ax<x> := PolynomialRing(Integers());
> f := x^5 + 343*x^4 + 49*x^3 + 343*x^2 + 7*x + 6;
> L := NumberField(f);
> Montes(L,7);
> P := L`PrimeIdeals[7,1];
> I := OMRepresentation(L,[L.1]);
```

(continues on next page)

(continued from previous page)

```
> I;
OM ideal of the field Number Field with defining polynomial x^5 + 343*x^4 +
49*x^3 + 343*x^2 + 7*x + 6 over the Rational Field
generated by [
$.1
]

> J := P^-2;
> I*J;
> I^2;
OM ideal of the field Number Field with defining polynomial x^5 + 343*x^4 +
49*x^3 + 343*x^2 + 7*x + 6 over the Rational Field
having the factorization [ <2, 1, 2>, <3, 1, 2> ]

> I/J;
OM ideal of the field Number Field with defining polynomial x^5 + 343*x^4 +
49*x^3 + 343*x^2 + 7*x + 6 over the Rational Field
having the factorization [ <2, 1, 1>, <3, 1, 1>, <7, 1, 2> ]

> I+J;
OM ideal of the field Number Field with defining polynomial x^5 + 343*x^4 +
49*x^3 + 343*x^2 + 7*x + 6 over the Rational Field
having the factorization  []
generated by [
1,
0
]
```

39.14.2 Ideal Predicates

IsOne(I): OMIdl -> BoolElt

Given an ideal I in OM representation returns whether the ideal is generated by the 1 element of the field.

IsZero(I): OMIdl -> BoolElt

Given an ideal I in OM representation returns whether the ideal contains only the 0 element of the field.

I **eq** J: OMIdl, OMIdl -> BoolElt

Given two ideals I and J in OM representation returns whether these ideals are the same.

a **in** I: RngElt, OMIdl -> BoolElt

Given an element a coercible into the field containing the ideal I in OM representation return whether a is contained in the ideal.

I **subset** J: OMIdl, OMIdl -> BoolElt

Given two ideals I and J in OM representation returns whether I is contained in J.

IsPrime(I): OMIdl -> BoolElt

Given an ideal I in OM representation returns whether only the 1 ideal and I contain I.

IsIntegral(I): OMIdl -> BoolElt

Given an ideal I in OM representation returns whether all elements in the ideal I are integral.

39.15 Ideal Operations

pIntegralBasis(I, p): OMIdl, RngElt -> SeqEnum

Given an ideal I in OM representation and a prime element p in the field containing I, compute a p-integral basis for I.

SIntegralBasis(I, S): OMIdl, SeqEnum -> SeqEnum

SIntegralBasis(I, S): RngOrdFracIdl, [RngIntElt] -> SeqEnum

SIntegralBasis(I, S): RngFunOrdIdl, [RngUPolElt] -> SeqEnum

Given an ideal I in either representation and a sequence S of primes in the field containing I, compute an S-integral basis of I for the given set of primes S.

Basis(I): OMIdl -> SeqEnum

```
HNF      : BoolElt                    Default: false
Separated: BoolElt                    Default: false
```

Given an ideal I in OM representation, return a basis for I.

If HNF is set to true a triangular basis in Hermite form is returned.

If Separated is set to true the basis will be returned as a sequence of numerators and denominators.

Example: p-Integral Basis of an OM Ideal (ex-a203ed)

```
> Ax<x> := PolynomialRing(Integers());
> f := x^4 + 12*x^3 + 54*x^2 + 108*x + 89;
> L := NumberField(f);
> p := 2;
> I := OMRepresentation(L,[L.1,p^12]);
> pIntegralBasis(I,p);
[
    1,
    L.1 + 1,
    1/2*(L.1^2 + 2*L.1 + 1),
    1/4*(L.1^3 + 3*L.1^2 + 3*L.1 + 1)
]
> pIntegralBasis(I,p:HNF:=true); // In HNF
[
    1,
    L.1,
    1/2*(L.1^2 + 1),
    1/4*(L.1^3 + L.1^2 + 3*L.1 + 3)
]
> Basis(I);
[
    1,
    L.1 + 1,
    1/2*(L.1^2 + 2*L.1 + 1),
    1/4*(L.1^3 + 3*L.1^2 + 3*L.1 + 1)
]
> Basis(I : HNF := true);
[
    1,
    L.1,
    1/2*(L.1^2 + 1),
    1/4*(L.1^3 + L.1^2 + 3*L.1 + 3)
]
```

TwoElement(I): OMIdl -> FldArithElt, FldArithElt

Given an ideal I in OM representation, return a, b such that $e = a * e_1 + b * e_2$ for some e_1, e_2 for all $e \in I$.

Norm(I): OMIdl -> RngElt

Given an ideal I in OM representation, compute the norm of I.

Valuation(alpha, P : parameters): FldArithElt, OMIdl -> RngIntElt, FldElt

Valuation(alpha, P : parameters): FldRatElt, OMIdl -> RngIntElt, FldElt

Valuation(alpha, P : parameters): RngIntElt, OMIdl -> RngIntElt, FldElt

Valuation(alpha, P : parameters): RngUPolElt, OMIdl -> RngIntElt, FldElt

```
RED    : BoolElt                    Default: false,
MoreSFL: BoolElt                    Default: false
```

Compute the P-valuation v of α at the prime ideal P.

Setting the parameter `MoreSFL` to `true` selects a single factor lifting algorithm. Setting the parameter `RED` to `true` returns also the class of α in $P^v/P^{(v+1)}$.

Valuation(I, P): OMIdl, OMIdl -> RngIntElt

Given ideals I and P in OM representation, return the valuation of I at P.

a mod P: FldArithElt, OMIdl -> FldArithElt

Reduction(a, P): FldArithElt, OMIdl -> FldArithElt

Reduction(a, P, m): FldArithElt, OMIdl, RngIntElt -> [FldArithElt]

Given an element a of the field containing the prime ideal P, which is in OM representation, return a' such that $a = a' + I$ and $a' \in P^0/P$.

If $m > 0$ is given then a sequence of length m of elements in P^0/P is returned representing the local expansion of a at P up to precision m.

Factorization(I): OMIdl -> SeqEnum

Factorisation(I): OMIdl -> SeqEnum

Given an ideal I in OM representation returns a sequence of tuples of primes P_i and exponents e_i such that $I = \prod_i P_i^{e_i}$.

Example: Operations on OM Ideals (ex-b5c67b)

```
> Ax<x> := PolynomialRing(Integers());
> f := x^5 + 343*x^4 + 49*x^3 + 343*x^2 + 7*x + 6;
> L := NumberField(f);
> I := OMRepresentation(L,[1/L.1^2,12]);
> I;
OM ideal of the field Number Field with defining polynomial x^5 + 343*x^4 +
49*x^3 + 343*x^2 + 7*x + 6 over the Rational Field
generated by [
1/36*(7*$.1^4 + 2395*$.1^3 - 1715*$.1^2 + 2107*$.1 - 2009),
12
]

> TwoElement(I);
1
1/36*(91*L.1^4 + 211*L.1^3 + 169*L.1^2 + 175*L.1 + 55)
> Norm(I);
1/36
> Factorization(I);
[
    <OM prime ideal over   2
```

(continues on next page)

(continued from previous page)

```
    of Number Field with defining polynomial x^5 + 343*x^4 + 49*x^3 + 343*x^2 +
    7*x + 6 over the Rational Field
    having residual degree   1
    and ramification index   1
    Last phi polynomial is   x, -2>,
    <OM prime ideal over   3
    of Number Field with defining polynomial x^5 + 343*x^4 + 49*x^3 + 343*x^2 +
    7*x + 6 over the Rational Field
    having residual degree   1
    and ramification index   1
    Last phi polynomial is   x, -2>
]
> Valuation(I, L`PrimeIdeals[2][1]);
-2
> Valuation(I, L`PrimeIdeals[3][1]);
-2
```

`ResidueField(I): OMIdl -> Fld`

Given an ideal I in OM representation returns the field P^0/P.

Example: Residual Degree and Ramification of OM Primes `(ex-7694e8)`

```
> Ax<x> := PolynomialRing(Integers());
> f := x^5 + 343*x^4 + 49*x^3 + 343*x^2 + 7*x + 6;
> L := NumberField(f);
> p := 7;
> Montes(L,p);
> L`PrimeIdeals[p];
[
    OM prime ideal over   7
    of Number Field with defining polynomial x^5 + 343*x^4 + 49*x^3 + 343*x^2 +
    7*x + 6 over the Rational Field
    having residual degree   1
    and ramification index   1
    Last phi polynomial is x + 6,
    OM prime ideal over   7
    of Number Field with defining polynomial x^5 + 343*x^4 + 49*x^3 + 343*x^2 +
    7*x + 6 over the Rational Field
    having residual degree   4
    and ramification index   1
    Last phi polynomial is x^4 + x^3 + x^2 + x + 1
]
> ResidueField($1[2]);
Finite field of size 7^4
```

39.16 Bibliography

- [Bai96] Georg Baier. Zum Round 4 Algorithmus. Master's thesis, Technische Universität Berlin, 1996. http://www.math.tu-berlin.de/~kant/publications/diplom/baier.ps.gz.
- [BF15] Karim Belabas and Eduardo Friedman. Computing the residue of the Dedekind zeta function. *Math. Comp.*, 84(291):357–369, 2015. URL: https://doi.org/10.1090/S0025-5718-2014-02843-3, doi:10.1090/S0025-5718-2014-02843-3.
- [Ber05] Daniel J. Bernstein. Factoring into coprimes in essentially linear time. *J. of Algorithms*, 54(1):1–30, 2005.

- [Bia] J. Biasse. “Number field sieve to compute Class groups.”
- [BH96] Yuri Bilu and Guillaume Hanrot. Solving Thue Equations of High Degree. *J. Number Th.*, 60:373–392, 1996.
- [BL94c] Johannes A. Buchmann and Hendrik W. Lenstra jr. Approximating rings of integers in number fields. *J. Théor. Nombres Bordx.*, 6(2):221–260, 1994.
- [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.
- [FJP97] C. Fieker, A. Jurk, and M. Pohst. On solving relative norm equations in algebraic number fields. *Math. Comput.*, 66(217):399–410, 1997.
- [Fie97] Claus Fieker. *Über relative Normgleichungen in algebraischen Zahlkörpern.* Dissertation, Technische Universität Berlin, 1997. http://www.math.tu-berlin.de/∼kant/publications/diss/diss_CF.ps.gz.
- [FF] C. Fieker, C. Friedrichs. “On reconstruction of algebraic numbers.” 285–296.
- [Fin84] Ulrich Fincke. *Ein Ellipsoidverfahren zur Lösung von Normgleichungen in algebraischen Zahlkörpern.* Dissertation, Heinrich-Heine-Universität Düsseldorf, 1984.
- [Fri97] Carsten Friedrichs. Berechnung relativer Ganzheitsbasen mit dem Round-2-Algorithmus. Master's thesis, Technische Universität Berlin, 1997. 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. http://www.math.tu-berlin.de/∼kant/publications/diss/diss_fried.pdf.gz.
- [GPP93] Istvan Gaál, Attila Petho, 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 Petho, 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.
- [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.
- [GMN13] Jordi Guàrdia, Jesús Montes, and Enric Nart. A New Computational Approach to Ideal Theory in Number Fields. *Foundations of Computational Mathematics*, 13(5):729–762, Oct 2013. URL: https://doi.org/10.1007/s10208-012-9137-5, doi:10.1007/s10208-012-9137-5.
- [GNP12] Jordi Guàrdia, Enric Nart, and Sebastian Pauli. Single-factor lifting and factorization of polynomials over local fields. *Journal of Symbolic Computation*, 47(11):1318 – 1346, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0747717112000363, doi:https://doi.org/10.1016/j.jsc.2012.03.001.
- [Hes96] Florian Heß. Zur Klassengruppenberechnung in algebraischen Zahlkörpern. Master's thesis, Technische Universität Berlin, 1996. 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.
- [Mul25] Erik Mulder. Fast square-free decomposition of integers using class groups. *Res. Number Theory*, 11(1):Paper No. 9, 21, 2025. URL: https://doi.org/10.1007/s40993-024-00585-8, doi:10.1007/s40993-024-00585-8.

- [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.
- [Poh] M. E. Pohst. "Computational Aspects of Kummer Theory." 259–272.
- [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.
- [Sta18] Hayden D. Stainsby. Triangular bases of integral closures. *J. Symbolic Comput.*, 87:140–175, 2018. URL: https://doi.org/10.1016/j.jsc.2017.07.012, doi:10.1016/j.jsc.2017.07.012.
- [Sut12] Nicole Sutherland. Efficient Computation of Maximal Orders of Radical (including Kummer) Extensions. *Journal of Symbolic Computation*, 47(5):552–567, 2012. doi:10.1016/j.jsc.2011.12.053.
- [Tra76] Barry M. Trager. Algebraic factoring and rational function integration. In R.D. Jenks, editor, *Proc. SYMSAC '76*, 196–208. ACM press, 1976.
- [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. 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.
- [KAN97] KANT Group. KANT V4. *J. Symbolic Comp.*, 24(3–4):267–383, 1997.
- [KAN00] KANT Group. The Number Theory Package KANT/KASH. http://www.math.tu-berlin.de/∼kant, 2000.

Chapter 40

Galois Groups and Automorphisms

This chapter deals with Galois groups and automorphism of number fields and several kinds of function fields. It also deals with computing the subfields of these kinds of fields. While these problems are closely related from a theoretical point of view (basically, everything is determined by the Galois group), as algorithmic problems they are quite different.

The first task, that of computing automorphisms of normal extensions of $\mathbb{Q}$ (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.

40.1 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): FldAlg -> [ Map ]`

```
Abelian          : BoolElt                    Default: false
AutomorphismGroup: Verbose                    Default: 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 [AK99, Klu97] will be employed. If F is not a normal extension, the automorphisms are obtained by a variation of the polynomial factorisation algorithm.

AutomorphismGroup(F): FldAlg -> GrpPerm, PowMap, Map

```
Abelian          : BoolElt                    Default: false
UseGalois        : BoolElt                    Default: false
AutomorphismGroup: Verbose                    Default: 3
```

Given an algebraic field F, that is either a simple normal extension of $\mathbb{Q}$ or simple abelian extension of $\mathbb{Q}$, 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 [AK99, Klu97] will be employed. If `UseGalois` is set, `GaloisGroup` will be called and the result will be derived from that. If F is not a normal extension of $\mathbb{Q}$ 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: Automorphisms of a Quartic Field (ex-7b9102)

We consider the extension obtained by adjoining a root of the irreducible polynomial $x^4 - 4x^2 + 1$ to $\mathbb{Q}$.

```
> 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): FldAlg, FldAlg -> GrpPerm, PowMap, Map

AutomorphismGroup(K, F): FldRat, FldAlg -> GrpPerm, PowMap, Map

AutomorphismGroup(K, F): FldAlg, FldRat -> GrpPerm, PowMap, Map

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 F over $\mathbb{Q}$ first.

`DecompositionGroup(p): RngIntElt -> GrpPerm`

`DecompositionGroup(p): RngOrdIdl -> GrpPerm`

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 $\mathbb{Q}$ an error will occur.

`RamificationGroup(p, i): RngOrdIdl, RngIntElt -> GrpPerm`

For an ideal p of the maximal order M 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 $\mathbb{Q}$ an error will occur.

`RamificationGroup(p): RngOrdIdl -> GrpPerm`

This is just an abbreviation for `RamificationGroup(p, 1)`.

`InertiaGroup(p): RngOrdIdl -> GrpPerm`

This is just an abbreviation for `RamificationGroup(p, 0)`.

`FixedField(K, U): FldAlg, GrpPerm -> FldNum, Map`

Given a normal field K over $\mathbb{Q}$ and a subgroup U of `AutomorphismGroup(K)`, this returns the largest subfield of K that is fixed by U.

This function is inverse to *FixedGroup*.

If K is not a normal extension of $\mathbb{Q}$ an error will occur.

`FixedField(K, S): FldAlg, [Map] -> FldAlg, Map`

Given a field K and a sequence of automorphisms of K, this returns the largest subfield of K that is fixed by the given automorphisms.

`FixedGroup(K, L): FldAlg, FldAlg -> GrpPerm`

Given a normal field K over $\mathbb{Q}$ 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 $\mathbb{Q}$ an error will occur.

`FixedGroup(K, L): FldAlg, [FldAlgElt] -> GrpPerm`

Given a normal field K over $\mathbb{Q}$ 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 $\mathbb{Q}$ an error will occur.

`FixedGroup(K, a): FldAlg, FldAlgElt -> GrpPerm`

Given a normal field K over $\mathbb{Q}$ 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 $\mathbb{Q}$ an error will occur.

`DecompositionField(p): RngOrdIdl -> FldNum, Map`

This is an abbreviation for `FixedField(K, DecompositionGroup(p))` where K is the number field of the order of p.

`RamificationField(p, i): RngOrdIdl, RngIntElt -> FldNum, Map`

This is an abbreviation for `FixedField(K, RamificationGroup(p, i))` where K is the number field of the order of p.

`RamificationField(p): RngOrdIdl -> FldNum, Map`

This is an abbreviation for `FixedField(K, RamificationGroup(p))` where K is the number field of the order of p.

`InertiaField(p): RngOrdIdl -> FldNum, Map`

This is an abbreviation for `FixedField(K, InertiaField(p))` where K is the number field of the order of p.

Example: Ramification Groups of a Normal Field (ex-3b0e83)

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;
```

(continues on next page)

(continued from previous page)

```
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 $.1^4 +
    80346384509631057182412*$.1^3 +
    22568355830378814326531151377362093966156930 22*$.\\
    1^2 + 279581809285547646905698973995584573657929 1605177\\
    3809455107173769804*$.1 +
    22077876856825539803853422635266440799754188013751 61428\\
    41147104301325760481728833650060994 over the Rational
Field
> Discriminant($1);
10700005925626216180895747020647047166414333000723923591882\\
57829873417638072117114945163507537844711544617147344227643\\
21408503489566949866295669400825222748660907808235401444104\\
29329493645714658394673579309893726532999745496689571082958\\
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);
```

$D3$ should be a subfield of $I3$, 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 $P3$ is totally ramified over $I3$:

```
> 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),
```

(continues on next page)

(continued from previous page)

```
    (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): FldNum, RngIntElt -> GrpPermElt

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: Frobenius Elements in a Galois Group (ex-87fbb9)

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)
```

40.2 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 by [Gei03, GK00] and recent work by Klüners and Fieker [FK14], Elsenhans [Els12, Els14, Els16], and Sutherland [Sut15].

For polynomials over $\mathbb{Z}, \mathbb{Q}$, number fields and global function fields and irreducible polynomials over function fields over $\mathbb{Q}$, 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 $\mathbb{Q}$ which we get by adjoining a root of f to $\mathbb{Q}$. 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): RngUPolElt[RngInt] -> GrpPerm, SeqEnum, GaloisData`

`GaloisGroup(f): RngUPolElt[FldRat] -> GrpPerm, SeqEnum, GaloisData`

`GaloisGroup(f): RngUPolElt[FldAlg] -> GrpPerm, SeqEnum, GaloisData`

`GaloisGroup(f): RngUPolElt[RngOrd] -> GrpPerm, SeqEnum, GaloisData`

```
Prime      : RngElt                    Default:
ShortOK    : BoolElt                   Default: false
Ring       : GaloisData                Default:
NextPrime  : UserProgram               Default:
GaloisGroup: Verbose                   Default: 5
Invariant  : Verbose                   Default: 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.

Along with the Galois group of a splitting field for f, the roots of f, scaled to be algebraic integers, in a local splitting field and a `GaloisData` structure are returned.

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`. If the `Prime` given ramifies and the polynomial is defined over the rationals, the computation will be done in an expensive ramified splitting field. If the polynomial is not defined over the rationals, an unramfied one larger than that given will be chosen.

If a `GaloisData` structure has been computed for this polynomial it can be provided in `Ring`. If `ShortOK` is set to `true` then the results of a descent using short cosets [Els14] will be assumed to be as reliable as a descent which did not use short cosets.

`IsEasySnAn(f): RngUPolElt -> RngIntElt`

```
Trials              : RngIntElt                Default: 50
AssumeIrreducibility: BoolElt                  Default: false
Rigorous            : BoolElt                  Default: true
```

Given a rational polynomial f, attempt some cheap tests to determine if the Galois group is easily determinable as S_n or A_n. Returns 1 if S_n is proven, 2 is A_n is proven, and 0 otherwise. The computation is done by factoring reductions modulo p and an inspection of the discriminant (if necessary). If `Rigorous` is set to `false`, the discriminant will not be computed.

`GaloisGroup(K): FldNum -> GrpPerm, SeqEnum, GaloisData`

`GaloisGroup(K): FldNum[FldRat] -> GrpPerm, SeqEnum, GaloisData`

GaloisGroup(K): FldNum[FldNum[FldRat]] -> GrpPerm, SeqEnum, GaloisData

Prime	: RngElt	Default:
ShortOK	: BoolElt	Default: false
Ring	: GaloisData	Default:
NextPrime	: UserProgram	Default:
Current	: BoolElt	Default: false
Subfields	: BoolElt	Default: true
Deflation	: BoolElt	Default: false
Old	: BoolElt	Default: false
Type	: MonStgElt	Default: "p-Adic"
Prec	: RngIntElt	Default: 20
Time	: BoolElt	Default: true
GaloisGroup	: Verbose	Default: 5
Invariant	: Verbose	Default: 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.

Along with the Galois group of a splitting field for K, the roots of the defining polynomial of K, scaled to be algebraic integers, in a local splitting field and a **GaloisData** structure are returned.

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**. If the **Prime** given ramifies in K and K is not an absolute number field then an unramfied one larger than that given will be chosen.

If a **GaloisData** structure has been computed for the defining polynomial it can be provided in **Ring**. If **Old** is set to **true** the older version of [Gei03, GK00] will be used for the computation.

If **Subfields** is set to **false** then the **Subfields** of K will not be used during the computation of the Galois group. If **Current** is set to **true** then the previously computed subfields of K only will be used in the Galois group computation and no more will be computed.

If **Deflation** is set to **true** and the defining polynomial is of the shape $g(X^d)$ for some $d > 1$, not all subfields of K will be computed. Instead, K will be viewed as a degree d radical extension of the subfield W given by g. Only the subfields of W will be computed.

If **Type** is set to **"Complex"** then roots of f in the complex field will be used rather than roots of f in a p-adic completion. If **Type** is set to **"Complex"**, the parameter **Prec** can be set to the minimum precision required in the roots of f.

If **ShortOK** is set to **true** then the results of a descent using short cosets [Els14] will be assumed to be as reliable as a descent which did not use short cosets. If **Time** is set to **true** then the time taken for lifting of roots will be stored in the attribute **Time** of the third return value.

GaloisProof(f, S): RngUPolElt, GaloisData -> BoolElt

GaloisProof(K, S): FldAlg, GaloisData -> BoolElt

Given the resulting **GaloisData** structure S of a (conditional) computation by **GaloisGroup** for either a polynomial f over the integers or an absolute number field K, try to find proofs for the conditional steps of the algorithm. The method employed here is to show that a suitable resolvent polynomial has a factor of a specific degree that can be used to differentiate between the possible groups.

GaloisRoot(f, i, S): RngUPolElt, RngIntElt, GaloisData -> RngElt

GaloisRoot(i, S): RngIntElt, GaloisData -> RngElt

```
Prec  : RngIntElt                    Default: 20
Bound : RngIntElt                    Default: 0
Scaled: BoolElt                      Default: true
```

Given a polynomial f (optional) and the resulting GaloisData S of a computation of a Galois group, return the ith root of f in the ordering obtained from the Galois process or compute the ith conjugate of the primitive element used during the computation, 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 $p^k > B$. If Scaled is set to false a root of f is returned. Otherwise the result is a scalar multiple of the root which is an algebraic integer.

GaloisGroupConjugation(S, T): GaloisData, GaloisData -> GrpPermElt

Given GaloisData obtained from two runs of GaloisGroup with the same polynomial but different primes, this function returns a permutation of the roots of S that induces an isomorphism of the splitting fields.

GaloisAutomorphismGroup(K): FldNum[FldRat] -> GrpPerm, Map

```
Galois: Tup                    Default: false
```

Given a number field K, this function returns the automorphism group of the field K as an abstract group and a map that maps group elements to field automorphisms. The computation is done by calling GaloisGroup and converting the result. If GaloisGroup has been called before, the result can be supplied via Galois. This function is also available via AutomorphismGroup with the parameter UseGalois.

Stauduhar(G, H, S, B): GrpPerm, GrpPerm, GaloisData, RngIntElt -> RngIntElt, GrpPermElt, BoolElt, RngSLPolElt

```
GaloisGroup    : Verbose                 Default: 5
Invariant      : Verbose                 Default: 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 these 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. If AlwaysTransform is set to true then a transformation will always be applied to the roots.

`IsInt(x, B, S): RngElt, RngIntElt, GaloisData -> BoolElt, RngElt`

Given an element x in the splitting field determined by the `GaloisData` structure S and a bound B on the complex absolute value, determine if there exists an element $y \in \mathbb{Z}$ or an extension of $\mathbb{Z}$ defined by the polynomial the Galois group is being computed for, 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: Galois Group of a Sextic Polynomial (ex-cc334e)

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)
```

40.2.1 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 Invariant Theory, 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 these 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): Rng, RngIntElt -> RngSLPol`

```
Global: BoolElt                    Default: false
```

Creates the ring of multivariate straight-line polynomials over the ring R with n indeterminates. If `Global` is set to `true` then an existing straight-line polynomial ring will be returned if one has previously constructed.

`Name(R, i): RngSLPol, RngIntElt -> RngSLPolElt`

`R . i: RngSLPol, RngIntElt -> RngSLPolElt`

Return the ith indeterminate of the SL-polynomial ring R.

ElementarySymmetricPolynomial(R, i): RngSLPol, RngIntElt -> RngSLPolElt
The i-th elementary symmetric polynomial of the SL-polynomial ring R.

BaseRing(R): RngSLPol -> Rng
CoefficientRing(R): RngSLPol -> Rng
Return the coefficient ring of the SL-polynomial ring R.

Rank(R): RngSLPol -> RngIntElt
Return the rank of the SL-polynomial ring R, ie the number of independent indeterminates over the coefficient ring.

SetEvaluationComparison(R, F, n): RngSLPol, FldFin, RngIntElt
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 comparison of SL-polynomials through their values at random evaluation points.

GetEvaluationComparison(R): RngSLPol -> FldFin, RngIntElt
Return the finite field and the number of random samples used to compare polynomials of the SL-polynomial ring R. If `SetEvaluationComparison` has not been called, the 1st return value will be `false` while the second is undefined.

InitializeEvaluation(R, S): RngSLPol, [RngElt]
Store the point to evaluate straight line polynomials with parent R to contain the elements of the sequence S whose length is the rank of R and whose elements are coercible into R.

x * y: RngSLPolElt, RngSLPolElt -> RngSLPolElt
x + y: RngSLPolElt, RngSLPolElt -> RngSLPolElt
x - y: RngSLPolElt, RngSLPolElt -> RngSLPolElt
- x: RngSLPolElt -> RngSLPolElt
Derivative(x, i): RngSLPolElt, RngIntElt -> RngSLPolElt
The ith partial derivative of the SL-polynomial x.

Evaluate(f): RngSLPolElt -> RngElt
Evaluate(f, S): RngSLPolElt, [RngElt] -> RngElt
Evaluate(f, S, m): RngSLPolElt, [RngElt], Map -> RngElt
Return the evaluation of the straight-line polynomial f at the elements of the sequence S where S contains the rank of the parent of f many elements coercible into the coefficient ring of f and if not given as an argument to this intrinsic is the sequence which was last input to *InitializeEvaluation*. The map m from the coefficient ring of f into the universe of S can be specified if required.

40.2.2 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 group is a subgroup of U - provided it was contained in G. This is achieved by evaluating a G-relative U-invariant polynomial $f \in \mathbb{Z}[x_1, \ldots, x_n]$ (or $f \in \mathbb{F}_q[t][x_1, \ldots, x_n]$ when the characteristic of the coefficient ring of the input polynomial is prime). 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): GrpPerm, GrpPerm -> RngSLPolElt

```
DoCost     : BoolElt                    Default: false
Worklevel  : RngIntElt                  Default: -1
GaloisGroup: Verbose                    Default: 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): GrpPerm, GrpPerm -> RngSLPolElt`

```
IsMaximal: BoolElt                  Default: false
Risk     : BoolElt                  Default: false
Invariant: Verbose                  Default: 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. Otherwise Magma will compute all minimal overgroups of H in G and call *GaloisGroupInvariant* for each of them. Finally, Magma will combine these invariants to a relative one.

The parameter `Risk` refers to an old version of the function and does not have an effect.

`CombineInvariants(R, G, H1, H2, H3): Rng, GrpPerm, Tup<GrpPerm, RngSLPolElt>, Tup<GrpPerm, RngSLPolElt>, GrpPerm -> RngSLPolElt`

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 over the ring R (usually $\mathbb{Z}$). The input for H_1 and H_2 consists of a tuple with two (or three) entries, the first specifying the actual subgroup, the second the G-relative H_i-invariant, and the optional third a Tschirnhaus transformation that should be done before the invariant is evaluated.

The 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): RngSLPolElt, GrpPermElt -> BoolElt`

```
Sign: BoolElt                       Default: false
```

For a multivariate polynomial F in straight-line representation and a permutation p this function tests if $F^p = F$ 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): RngSLPolElt, RngIntElt -> RngIntElt`

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 $|x_i| \leq B$. This returns a bound for the size of an evaluation of I.

`Bound(I, B): RngSLPolElt, RngElt -> RngElt`

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.

`PrettyPrintInvariant(I): RngSLPolElt -> MonStgElt`

Prints the straight-line polynomial I into a string. The printing shows the way the polynomial is stored internally. E.g. the reuse of subexpressions gets visible.

40.2.3 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): FldNum, GrpPerm -> RngUPolElt, RngSLPolElt`

`GaloisSubgroup(S, U): GaloisData, GrpPerm -> RngUPolElt, RngSLPolElt`

`GaloisSubgroup(f, U): RngUPolElt, GrpPerm -> RngUPolElt, RngSLPolElt`

`Invariant: Verbose` Default: 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 and return also a G-relative U-invariant.

`GaloisSibling(S, phi): GaloisData, Map -> RngUPolElt, RngSLPolElt, RngUPolElt`

Given a map φ with domain the Galois group associated to S and a transitive permutation group as image, this function returns a resolvent with Galois group $\mathrm{im}(\varphi)$. It also returns the relative invariant and the transformation used.

`RelativeResolvent(S, U): GaloisData, GrpPerm -> RngUPolElt, RngSLPolElt, RngUPolElt`

Given a subgroup U of the Galois group G associated to S, this function returns a relative resolvent for the groups $U < G$. It also returns the relative invariant and the transformation used.

`GaloisQuotient(K, Q): FldNum, GrpPerm -> SeqEnum[RngUPolElt]`

`GaloisQuotient(f, Q): RngUPolElt, GrpPerm -> SeqEnum[RngUPolElt]`

`GaloisQuotient(S, Q): GaloisData, GrpPerm -> SeqEnum[RngUPolElt]`

`Invariant: Verbose` Default: 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. The defining polynomials of these subfields are returned. 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): GaloisData, [GrpPerm] -> FldNum, [Tup<RngSLPolElt, RngUPolElt, [GrpPermElt]>], UserProgram, UserProgram

```
Risk       : BoolElt                          Default: false
MinBound   : RngIntElt                        Default: 1
MaxBound   : RngIntElt                        Default: Infinity()
Inv        : [RngSLPolElt]                    Default: false
GaloisTower: Verbose                          Default: 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 $\mathbb{Q}, \mathbb{F}_q[t]$ or absolute extensions of $\mathbb{Q}$.

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 which is an extension of the fixed field of the third group etc. 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): RngUPolElt -> FldNum, [FldNumElt], GrpPerm, [[FldNumElt]]

```
Galois : Tup<GrpPerm, [RngElt], GaloisData>   Default:
Roots  : BoolElt                              Default: true
AllAuto: BoolElt                              Default: false
```

(continues on next page)

(continued from previous page)

```
Stab    : BoolElt                                          Default: true
Chain   : [GrpPerm]                                        Default: false
Inv     : [RngSLPolElt]                                    Default: false
Name    : MonStgElt                                        Default: false
```

For a polynomial f in $\mathbb{Z}[t]$, $\mathbb{Q}[t]$ or over an absolute number field or $\mathbb{F}_q[t]$, 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 $\{1\}$, $\{1,2\}$, ..., $\{1,\ldots,n\}$. 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: Galois Splitting Field of a Cubic (ex-4f61da)

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];
```

(continues on next page)

(continued from previous page)

```
[
    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
    ],
```

(continues on next page)

(continued from previous page)

```
    [
        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 ${}_{10}T_8$ 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 -
```

(continues on next page)

(continued from previous page)

```
    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 $\mathbb{Q}[x_1, \ldots, x_{10}]/I$ for some suitable ideal I also known as the Galois ideal. On the other hand, by tensoring with some p-adic complection $\mathbb{Q}_p$ 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 $\mathbb{Z}[x_1, \ldots, x_{10}]$ 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-transformation 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.

40.2.4 Solvability by Radicals

For a polynomial $f \in \mathbb{Z}[t]$ 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 a representation.

`SolveByRadicals(f): RngUPolElt -> FldNum, [FldNumElt], [FldNumElt]`

```
Prime      : RngIntElt                                    Default: false
Name       : MonStgElt                                    Default: false
Galois     : Tup<GrpPerm, [RngElt], GaloisData>           Default:
UseZeta_p  : BoolElt                                      Default: false
MaxBound   : RngIntElt                                    Default:␣
↪Infinity()
GaloisTower: Verbose                                      Default: 3
```

For a polynomial $f \in \mathbb{Z}[x], \mathbb{F}_q[t][x]$ 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 the `Prime` given ramifies in

the extensions defined by f then an unramfied one larger than that given will be chosen.

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. Especially 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): FldNum, FldNumElt, RngElt -> FldNum, [FldNumElt], [FldNumElt]`

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: Solving a Polynomial by Radicals `(ex-f5133d)`

```
> 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 $x^n - a$.

40.2.5 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): RngUPolElt -> Mtrx, GaloisData`

```
Proof       : BoolElt                                          Default: true
Galois      : Tup<GrpPerm, [RngElt], GaloisData>               Default:
UseAction   : BoolElt                                          Default:␣
↪false
UseLLL      : BoolElt                                          Default: true
Power       : RngIntElt                                        Default: 1
kMax        : RngIntElt                                        Default:␣
↪Infinity()
LogLambdaMax: RngIntElt                                        Default:␣
↪Infinity()
```

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): RngUPolElt, [RngSLPolElt] -> Mtrx, GaloisData`

```
Proof       : BoolElt                                          Default: true
Galois      : Tup<GrpPerm, [RngElt], GaloisData>               Default:
UseAction   : BoolElt                                          Default:␣
↪false
UseLLL      : BoolElt                                          Default: true
Power       : RngIntElt                                        Default: 1
kMax        : RngIntElt                                        Default:␣
↪Infinity()
LogLambdaMax: RngIntElt                                        Default:␣
↪Infinity()
```

Let f be an integral monic polynomial and $\alpha_1, \ldots, \alpha_n$ 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 $\mathbb{Q}[x_1, \ldots, x_n]/J$ 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): RngUPolElt, RngSLPolElt -> BoolElt`

```
Galois: Tup<GrpPerm, [RngElt], GaloisData>                     Default:
kMax  : RngIntElt                                              Default: Infinity()
```

Let f be an integral monic polynomial and $\alpha_1, \ldots, \alpha_n$ 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 $\mathbb{Q}[x_1, \ldots, x_n]/J$ 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: Linear Relations among Roots `(ex-2acfb0)`

The following example originates in a paper [BDE+] where, among other things, polynomials are constructed whose roots have a maximal number of linear dependencies. (This example is not extremal.)

```
> ST := ShephardTodd(8);
> R := InvariantRing(ST);
> 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
]
> res := Resultant(p[1]-2, p[2]-3, 2);
> f4  := Polynomial(Rationals(), UnivariatePolynomial(res));
> bool, f := IsPower(f4, 4);
> bool, f;
true
27/4*x^24 - 135*x^16 + 405*x^12 - 405*x^8 + 162*x^4 - 1
> G, R, S := GaloisGroup(f);
> #G;
192
> rel := LinearRelations(f : Galois := <G, R, S>);
> #rel;
20
```

There are 20 linear relations between the 24 roots, that is, they span a vector space of dimension 4 over $\mathbb{Q}$. We demonstrate how to check one of the relations.

```
> v := rel[3];
> v;
[ 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 := &+ [v[i]*IR.i : i in [1..24]];
> r;
(x7 + ((-1 * x4) + (-1 * x6)))
> VerifyRelation(f, r : Galois := <G, R, S>);
true
```

Next we give a primitive element in the splitting field (of degree 192 over $\mathbb{Q}$) such that the $\mathbb{Q}$-vector space spanned by all 192 conjugates has dimension 4. The element is defined as a linear combination of the 24 roots (most random choices work). We show that the element is primitive by checking that the 192 p-adic conjugates are distinct.

```
> I := &+ [ (e[n]-1)*IR.n : n in [1..24]] where e is Eltseq(Random(Sym(24)));
> Iorbit := [Apply(g, I) : g in G];
> #{Evaluate(i, R) : i in Iorbit};
192
```

Thus the 192 conjugates are distinct, and by construction they lie in the linear span of the $alpha_i$, which has dimension 4 over $\mathbb{Q}$ as shown. It is possible to check this directly (i.e. p-adically, using the Galois data).

```
> time #LinearRelations(f, Iorbit : Galois := <G, R, S>, Proof := false);
188
Time: 645.450
```

One expects most primitive elements of the field not to share this property. We define one by choosing a polynomial $I = x_1 + x_n^2$ that has trivial stabilizer in G.

```
> exists(n){n : n in [1..24] | #Orbit(G, [1,n]) eq 192}; n;
true
2
> I := IR.1 + IR.n^2;
> Iorbit := [Apply(g, I) : g in G];
> #{Evaluate(i, R) : i in Iorbit};
192
> time #LinearRelations(f, Iorbit : Galois := <G, R, S>, Proof := false);
182
Time: 816.120
```

40.2.6 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.

40.3 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 similarly there is no limit on the degrees of the field.

The algorithms used are Klüner's method as presented in [Klu95, Klu97, Klu98, KP97] and the newer method of Klüners, van Hoeij, and Novocin [vHKN11].

Subfields(K, n): FldAlg, RngIntElt -> [< FldAlg, Hom >]

```
Subfields: Verbose                    Default: 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): FldAlg -> [< FldAlg, Hom >]

```
Al       : MonStgElt                  Default: "Default"
Current  : BoolElt                    Default: false
Proof    : RngIntElt                  Default: 1
Subfields: Verbose                    Default: 4
```

Given an algebraic field K, this function returns a sequence of pairs (2-tuples) containing the subfields of K (except $\mathbb{Q}$) 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 $\mathbb{Q}$, 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.

40.3.1 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): FldNum -> SubFldLat

```
Subfields: Verbose                    Default: 4
```

The lattice of subfields of an absolute number field K.

L: SubFldLat -> RngIntElt

The number of fields in the lattice L.

Representative(L): SubFldLat -> SubFldLatElt

Rep(L): SubFldLat -> SubFldLatElt

Bottom(L): SubFldLat -> SubFldLatElt

The bottom element of the subfield lattice L (this corresponds to $\mathbb{Q}$).

Top(L): SubFldLat -> SubFldLatElt

The top element of the subfield lattice L (this corresponds to the original number field).

Random(L): SubFldLat -> SubFldLatElt

A random element of the subfield lattice L.

L ! n: SubFldLat, RngIntElt -> SubFldLatElt

L[n]: SubFldLat, RngIntElt -> SubFldLatElt

The n-th element of the subfield lattice L.

NumberField(e): SubFldLatElt -> FldNum

The number field corresponding to the given subfield lattice element e.

EmbeddingMap(e): SubFldLatElt -> Map

The mapping from `NumberField(e)` into the top number field of the subfield lattice.

Degree(e): SubFldLatElt -> RngIntElt

The (absolute) degree of the number field corresponding to the subfield lattice element e.

e eq f: SubFldLatElt, SubFldLatElt -> BoolElt

Returns `true` if and only if the subfield lattice elements e and f are equal.

e subset f: SubFldLatElt, SubFldLatElt -> BoolElt

Returns `true` if and only if e is a subfield of f.

e * f: SubFldLatElt, SubFldLatElt -> SubFldLatElt

The smallest field containing both e and f.

e meet f: SubFldLatElt, SubFldLatElt -> SubFldLatElt

The intersection of e and f. This is the largest field common to both of them.

&meet S: [SubFldLatElt] -> SubFldLatElt

The intersection of the subfields in the sequence S.

`MaximalSubfields(e): SubFldLatElt -> [ SubFldLatElt ]`

The sequence of maximal subfield lattice elements contained in e.

`MinimalOverfields(e): SubFldLatElt -> [ SubFldLatElt ]`

The sequence of minimal subfield lattice elements containing e.

Example: Subfield Lattice of an Octic Field (ex-c7abdb)

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
```

40.4 Galois Cohomology

Magma has some rudimentary functions to aid computations in Galois cohomology of number fields.

`Hilbert90(a, M): FldNumElt, Map[FldNum, FldNum] -> FldNumElt`

```
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): {RngOrdIdl}, GrpPerm -> {1}`

```
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): {}, UserProgram -> BoolElt, UserProgram`

```
Sub       : GrpPerm                      Default: false
Cohomology: Verbose                      Default: 2
```

For a cohomology process C as created by *SUnitCohomologyProcess* and a 2-cocycle $l : U \times U \to k$ 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 cocycle easily.

As a fixed cocycle l assumes only finitely many values, we can consider it as a cocycle 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): RngOrdFracIdl, UserProgram -> BoolElt, UserProgram, [RngOrdIdl]`

```
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-cocycle, 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 cocycle 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.

40.5 Automorphisms of Fields

Magma now (from V2.29) contains a new structure for field automorphisms. A field automorphism is of type `FldAut`. Also supports the swap automorphism of $F \times F$, when constructed as an associative algebra. `DirectSum(Algebra(F,F), Algebra(F,F));`

40.5.1 Creation of Field Automorphisms

`FieldAutomorphism(F, g): Fld, GrpPermElt -> FldAut`

`FieldAutomorphism(F, g): AlgAss[Fld], GrpPermElt -> FldAut`

Given an element g of a permutation group G, satisfying $G \simeq \mathrm{Aut}(F)$, returns an element $\alpha \in \mathrm{Aut}(F)$ corresponding to g.

`FieldAutomorphism(F, a): Fld, Map[Fld,Fld] -> FldAut`

`FieldAutomorphism(F, a): Fld, Intrinsic -> FldAut`

`FieldAutomorphism(F, a): Fld, UserProgram -> FldAut`

`FieldAutomorphism(F, a): AlgAss[Fld], Map[AlgAss[Fld],AlgAss[Fld]] -> FldAut`

Given a function, an intrinsic, or a map between fields a describing an automorphism $a : F \to F$, returns the corresponding field automorphism.

`IdentityAutomorphism(F): Fld -> FldAut`

The identity automorphism of the field F.

`FieldAutomorphism(K): AlgAss[Fld] -> FldAut`

If $K = F \times F$, return the swap involution.

`ChangeRing(a, K): FldAut, Fld -> FldAut`

`ChangeRing(a, K): FldAut, AlgAss[Fld] -> FldAut`

If $F \simeq K$, return the element of $\mathrm{Aut}(K)$ corresponding to $a \in \mathrm{Aut}(F)$.

40.5.2 Properties of Field Automorphisms

`BaseField(a): FldAut -> Fld`

The field F for which $a \in \mathrm{Aut}(F)$.

`Order(a): FldAut -> RngIntElt`

The order of a.

`FixedField(a): FldAut -> Fld`

The fixed field of a.

`Automorphism(a): FldAut -> Map[Fld, Fld]`

The map defined by $a : F \to F$.

`FixedFieldSymbol(a, P): FldAut, RngOrdIdl -> RngIntElt`

`FixedFieldSymbol(a, P): FldAut, RngInt -> RngIntElt`

Returns $-1, 0$ or 1 if P is inert, ramified, or split, respectively in the fixed field of a.

40.5.3 Predicates on Field Automorphisms

`IsIdentity(a): FldAut -> BoolElt`

Returns `true` iff a is the identity automorphism.

40.5.4 Arithmetic of Field Automorphisms

`a * b: FldAut, FldAut -> FldAut`

The automorphism given by $a \circ b$, namely $x \mapsto a(b(x))$.

`a ^ n: FldAut, RngIntElt -> FldAut`

The automorphism a^n.

`Inverse(a): FldAut -> FldAut`

The automorphism a^{-1}.

`a eq b: FldAut, FldAut -> BoolElt`

Returns `true` iff a and b are automorphisms of the same field F, and $a = b$.

`x @ a: FldElt, FldAut -> FldElt`

`x @ a: RngOrdElt, FldAut -> RngOrdElt`

`x @ a: RngIntElt, FldAut -> RngIntElt`

`x @ a: AlgAssElt[Fld], FldAut -> AlgAssElt[Fld]`

Evaluate $a(x)$.

`v @ a: ModTupFldElt[Fld], FldAut -> ModTupFldElt[Fld]`

Given a vector $v = (x_1, ..., x_n)$ in an F-vector space V, apply a to each coordinate, to get $a(v) = (a(x_1), \ldots, a(x_n))$.

`M @ a: ModMatFldElt[Fld], FldAut -> ModMatFldElt[Fld]`

`M @ a: AlgMatElt[Fld], FldAut -> AlgMatElt[Fld]`

`M @ a: GrpMatElt[Fld], FldAut -> GrpMatElt[Fld]`

`M @ a: AlgMatElt[AlgAss[Fld]], FldAut -> AlgMatElt[AlgAss[Fld]]`

Given a matrix $M = (x_{ij})$ with entries in F, apply a to each coordinate, to get $a(M) = (a(x_{ij}))$.

`I @ a: RngOrdFracIdl[FldOrd], FldAut -> RngOrdFracIdl[FldOrd]`

`I @ a: RngInt, FldAut -> RngInt`

`I @ a: RngIntFracIdl, FldAut -> RngIntFracIdl`

Given a fraction ideal I of an order R, and an automorphism a of the fraction field of R, return the fractional ideal $a(I)$.

`Trace(P, a): RngOrdFracIdl, FldAut -> RngOrdFracIdl`

`Trace(P, a): RngIntFracIdl, FldAut -> RngIntFracIdl`

```
Mult: RngIntElt                    Default: Order(a)
```

The image of the map $x \mapsto \sum_{j=0}^{m-1} \sum a^j(x)$ on P, where m is determined by `Mult`.

`Norm(P, a): RngOrdFracIdl, FldAut -> RngOrdFracIdl`

`Norm(P, a): RngIntFracIdl, FldAut -> RngIntFracIdl`

```
Mult: RngIntElt                    Default: Order(a)
```

The image of the map $x \mapsto \prod_{j=0}^{m-1} \sum a^j(x)$ on P, where m is determined by `Mult`.

40.6 Bibliography

- [AK99] Vincenzo Acciaro and Jürgen Klüners. Computing Automorphisms of Abelian Number Fields. *Math. Comp.*, 68(227):1179–1186, 1999.
- [BDE+] N. Berry, A. Dubickas, N. Elkies, B. Poonen, C. 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.
- [Els12] A.-S. Elsenhans. Invariants for the computation of intransitive and transitive Galois groups. *Journal of Symbolic Computation*, 47:315–326, 2012.
- [Els14] A.-S. Elsenhans. A Note on Short Cosets. *Experimental Mathematics*, 23:411–413, 2014. doi:10.1080/10586458.2014.922908.
- [Els16] Andreas-Stephan Elsenhans. Improved methods for the construction of relative invariants for permutation groups. To appear in Journal of Symbolic Computation, Available online since 5 February 2016, 2016. doi:10.1016/j.jsc.2016.02.005.

- [FK14] C. Fieker and J. Klüners. Computation of Galois Groups of Rational Polynomials. *London Mathematical Society Journal of Computation and Mathematics*, 17(1):141 – 158, 2014. http://arxiv.org/abs/1211.3588.
- [Gei03] Katharina Geißler. *Berechnung von Galoisgruppen über Zahl- und Funktionenkörpern.* PhD Thesis, TU-Berlin, 2003. available at http://www.math.tu-berlin.de/ kant/publications/diss/geissler.pdf.
- [GK00] Katharina Geißler and Jürgen Klüners. Galois Group computation for Rational Polynomials. *J. Symbolic Comp.*, 30(6):653–674, 2000.
- [Klu95] Jürgen Klüners. Über die Berechnung von Teilkörpern algebraischer Zahlkörper. Master's thesis, Technische Universität Berlin, 1995. http://www.math.tu-berlin.de/∼kant/publications/diplom/klueners.ps.gz.
- [Klu97] Jürgen Klüners. *Über die Berechnung von Automorphismen und Teilkörpern algebraischer Zahlkörper.* Dissertation, Technische Universität Berlin, 1997. http://www.math.tu-berlin.de/∼kant/publications/diss/diss_jk.ps.gz.
- [Klu98] 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.
- [Sut15] Nicole Sutherland. Computing Galois Groups of Polynomials (especially over Function Fields of Prime Characteristic). *Journal of Symbolic Computation*, 71:73–97, 2015. doi:10.1016/j.jsc.2014.09.043.
- [vHKN11] M. van Hoeij, J. Klüners, and A. Novocin. Generating Subfields. In Anton Leykin, editor, *Proceedings ISSAC 2011.* 2011.

Chapter 41

Class Field Theory

41.1 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 *Class Field Theory for Global Function Fields* and for the case of p-adic local fields, Section Class Field Theory.

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.

41.1.1 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 $o = \mathbb{Z}_k$ 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 $\mathbb{C}$ 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. Furthermore, 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.

41.1.2 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 $k := \mathbb{Q}(\alpha)$ with $\alpha^3 + \alpha^2 + 3\alpha - 6 = 0$ with class group C_4.

Example: Hilbert Class Field of a Cubic Field (ex-546938)

```
> 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);
```

(continues on next page)

(continued from previous page)

```
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 := 0!!i;
> IsPrincipal(I);
true
> f,g := IsPrincipal(I);
> g;
[[219349778867847403 5456, -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:
```

(continues on next page)

(continued from previous page)

```
    [1, 0, 0]
[ 4, 4 ]
```

The second return value denotes the signature of the field as an extension of $\mathbb{Q}$.

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 remaining part collapses in `aK`.

41.2 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.

41.2.1 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* is relevant in this context.

RayClassGroup(I): RngOrdIdl -> GrpAb, Map

RayClassGroup(I, T): RngOrdIdl, SeqEnum[RngIntElt] -> GrpAb, Map

```
Al: MonStgElt                    Default: "Extension"
```

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.

Setting `Al` to `Extension` uses a mixture of Pauli's approach following Hasse ([HPP97, Pau96]) and Cohen's method ([Coh00, CDO, CDO98]). Setting `Al` to `Relations` uses a modification of the relation matrix to determine the ray class group. Currently, the second approach performs better in large examples.

`RayClassGroup(D): DivNumElt -> GrpAb, Map`

`RayClassGroup(P): PlcNumElt -> GrpAb, Map`

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 ([HPP97, Pau96]) and Cohen's method ([Coh00, CDO, CDO98]).

Example: Growth of Ray Class Groups `(ex-0ba809)`

Some ray class groups are computed below. The example merely illustrates the fact that ray class groups tend to grow if their defining module 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
```

(continues on next page)

(continued from previous page)

```
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): RngOrdIdl -> GrpAb, Map

RayResidueRing(I, T): RngOrdIdl, SeqEnum[RngIntElt] -> GrpAb, Map

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 $\mathrm{mod}^*(I, T)$. 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): DivNumElt -> GrpAb, Map

RayResidueRing(P): PlcNumElt -> GrpAb, Map

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.

41.2.2 Selmer Groups

Let S be a finite set of prime ideals in a number field K. For an integer p, the p-Selmer group of S is defined as

$$K_p(S) := \{x \in K^\times/(K^\times)^p \mid v_Q(x) = 0 \bmod p \ \forall Q \notin S\}$$

$K_p(S)$ is a finite abelian group of exponent p.

pSelmerGroup(p, S): RngIntElt, { RngOrdIdl } -> GrpAb, Map

```
Integral: BoolElt                    Default: true
Nice    : BoolElt                    Default: true
Raw     : BoolElt                    Default: false
```

For a prime integer p and a set of prime ideals S in a number field K, the function returns the p-Selmer group of S as an abstract group G, together with a map m from K to G. The map comes with an inverse.

In principle, the domain of m is the set of x in K satisfying the condition in the definition of $K_p(S)$ above. When $m(x)$ is invoked for x in K, it is assumed without checking that x satisfies the condition. If x does not, either a runtime error occurs, or the map returns a random element of G. (Checking the condition would require a far more expensive computation. The algorithm identifies the class of x in $K_p(S)$ by computing the multiplicative orders of residues of x, and of the group generators, modulo some unrelated primes.)

The role of the optional parameters is as follows. 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.

Example: 3-Selmer Group of a Real Quadratic Field (ex-2ae671)

We compute the 3-Selmer group of $\mathbb{Q}(\sqrt{10})$ 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  1  0  1  0  3  0  0)
> PowerProduct(B, $1);
396/1*m.1 + 99/1*m.2
```

41.2.3 Maps

InducedMap(m1, m2, h, c): Map, Map, Map, RngIntElt -> Map

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(m_1(G_1.x)) @@ m_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): Map, Map, RngIntElt -> Map

An abbreviation for `InducedMap(r, r, h, c)`.

Example: Induced Map on Ray Class Groups (ex-110f86)

Consider a "large" ray class group over $k := \mathbb{Q}[\sqrt{10}, \zeta_{16}]$

```
> k := NumberField([Polynomial([-10, 0, 1]), CyclotomicPolynomial(16)]);
> k := OptimizedRepresentation(AbsoluteField(k));
> o := MaximalOrder(k);
> ClassGroup(o : Proof := "GRH");
Abelian Group of order 1
Mapping from: Abelian Group of order 1 to Set of ideals of o
> IndependentUnits(o);
Abelian Group isomorphic to Z/16 + Z (7 copies)
Defined on 8 generators
```

(continues on next page)

(continued from previous page)

```
Relations:
    16*$.1 = 0
Mapping from: Abelian Group isomorphic to Z/16 + Z (7 copies)
Defined on 8 generators
Relations:
    16*$.1 = 0 to RngOrd: o
> SetOrderUnitsAreFundamental(o);
> p := &* [113, 193, 241];
> r, mr := RayClassGroup(p*o);
> #r; Ngens(r);
170613117637701990523621885614354740012596396318100496286167859 2\\
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: 1.080
```

In contrast, using `InducedAutomorphism`:

```
> time InducedAutomorphism(mr, autk[2], p);
Time: 1.010
```

Now we increase p:

```
> p *:= 257*337;
> r, mr := RayClassGroup(p*o);
> #r; Ngens(r);
383174842075502327821254012562863503503880824795585976926638885 1\\
825941987170813422862370672745342599097489263347018928299704320 0\\
0000000
42
> time m1 := hom<r -> r | [ autk[2](mr(r.i))@@ mr : i in [1..Ngens(r)]]>;
Time: 2.950
> time InducedAutomorphism(mr, autk[2], p);
Time: 6.200
```

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.

41.2.4 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): Map -> FldAb

AbelianExtension(m): Map -> FldAb

RayClassField(m, I, T): Map, RngOrdIdl, [RngIntElt] -> FldAb

AbelianExtension(m, I, T): Map, RngOrdIdl, [RngIntElt] -> FldAb

RayClassField(m, I): Map, RngOrdIdl -> FldAb

AbelianExtension(m, I): Map, RngOrdIdl -> FldAb

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 a 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.

Maps commonly given as m are class group maps, ray class group maps, and the composition of the inverse of the canonical map from a ray class group into the quotient of that ray class group by a subgroup and the ray class group map.

AbelianExtension(I): RngOrdIdl -> FldAb

Creates the full ray class field modulo the ideal I.

RayClassField(D): DivNumElt -> FldAb

RayClassField(P): PlcNumElt -> FldAb

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): Map, RngIntElt -> FldAb

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: Cyclotomic Field as a Class Field (ex-cfc4e4)

The abelian extensions of $\mathbb{Q}$ 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 $\mathbb{Z}$ in Magma, we must work in a degree 1 extension of $\mathbb{Q}$:

```
> x := ext<Rationals()|>.1;
> Q := ext<Rationals()| x-1 :DoLinearExtension>;
> M := MaximalOrder(Q);
```

The ray class group that defines $\mathbb{Q}(\zeta_{12})$ is defined $\bmod(12, \infty)$ where ∞ is the unique infinite place of $\mathbb{Q}$. There are at least two ways of looking at this: First, since $Q(\zeta_{12})$ is a totally complex field, the infinite place of $\mathbb{Q}$ must ramify (by convention: $\mathbb{C}$ is ramified over $\mathbb{R}$), so we must include the infinite place in the definition of the ray class group.

Secondly, the ray class group $\bmod(12)$ without the infinite place is too small. We know $\phi(12) = 4$, $(\mathbb{Z}/12\mathbb{Z})^* = \{1, 5, 7, 11\} = \{\pm 1, \pm 5\}$. 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);
```

(continues on next page)

(continued from previous page)

```
> 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]
```

(continues on next page)

(continued from previous page)

```
[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): RngOrdIdl, [RngIntElt] -> FldAb

Creates the full ray class field modulo the ideal I and the infinite places in P.

HilbertClassField(K): FldAlg -> FldAb

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): RngOrd -> FldAb

MaximalAbelianSubfield(F): FldOrd -> FldAb

MaximalAbelianSubfield(K): FldNum -> FldAb

```
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): FldAlg -> FldAb

AbelianExtension(M): RngOrd -> FldAb

```
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: Hilbert Class Field of a Sextic Field (ex-1fc01e)

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 : Proof := "GRH");
> a;
Abelian Group isomorphic to Z/3 + Z/3
```

(continues on next page)

(continued from previous page)

```
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: 2.200
```

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: 0.170
> 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.

41.2.5 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: FldAb, FldAb -> BoolElt`

Gives two abelian extensions with the same base field, decide if they are the same.

`A subset B: FldAb, FldAb -> BoolElt`

Gives two abelian extensions with the same base field, decide if they are contained in each other.

`A * B: FldAb, FldAb -> FldAb`

Given two abelian extensions with the same base field, find the smallest abelian extension containing both.

`A meet B: FldAb, FldAb -> FldAb`

Given two abelian extensions with the same base field, find the largest common subfield.

41.3 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, $g := \langle$`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 $\mathbb{Q}$ 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.

41.3.1 Predicates

`IsAbelian(A): FldAb -> BoolElt`

```
All : BoolElt                    Default: false
Over: [Map]                      Default: []
```

Returns `true` if and only if the abelian extension A is abelian over k_0.

`IsNormal(A): FldAb -> BoolElt`

```
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): FldAb -> BoolElt`

```
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.

41.3.2 Constructions

`GenusField(A): FldAb -> FldAb`

```
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): FldAb -> ModTupRng`

For A such that A is normal over $\mathbb{Q}$ 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): FldAb -> []`

```
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): FldAb, GrpAb -> FldAb`

`FixedField(A, U): FldAb, GrpAb -> FldAb`

```
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): FldAb -> ModGrp, Map, Map, Map`

For an abelian extension A defined over some normal field k/Q, compute the cohomology module (see Chapter Cohomology and Extensions). 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.

41.4 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): FldAb -> RngOrd`

```
ClassField: Verbose                  Default: 5
```

Given an abelian extension A of a number field, using the algorithm of Fieker ([Coh00, Fie00]) 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): FldAb -> FldNum`

Converts the abelian extension A into a number field. This is equivalent to `NumberField(EquationOrder(A))`.

`MaximalOrder(A): FldAb -> RngOrd`

```
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): FldAb -> [RngOrd]`

```
ClassField: Verbose                    Default: 5
```

A list of relative extensions is determined. One extension per cyclic factor is computed.

`Generators(A): FldAb -> [ ], [ ], [ ]`

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.

41.4.1 Character Theory

Given a Hecke character ψ of a number field K, one can compute an associated abelian extension L/K, and vice-versa. The same is true for Dirichlet characters over the rationals. The number field K must be given as an absolute field. See Chapter *Dirichlet and Hecke Characters* for more about Dirichlet and Hecke characters.

`AbelianExtension(psi): GrpHecke -> FldAb`

`AbelianExtension(psi): GrpHeckeElt -> FldAb`

`AbelianExtension(chi): GrpDrchNFElt -> FldAb`

`AbelianExtension(chi): GrpDrchElt -> FldAb`

Given a Hecke character group (or a generator if cyclic) over a number field K, or a Dirichlet character over the rationals (possibly represented as a number field), compute the corresponding Abelian field.

`HeckeCharacterGroup(L): FldNum -> GrpHecke`

Given an abelian relative extension L/K where K is absolute, compute the corresponding Hecke character group over K.

`HeckeCharacterGroup(A): FldAb -> GrpHecke`

Given an abelian field, compute the corresponding Hecke character group.

Example: Hecke Characters of a Class Field `(ex-d88d5d)`

We construct a cyclic cubic extension of a quartic field (of class number 2, it happens), and compute the corresponding Hecke character group, then check that the process inverts correctly. Finally we check the L-series of a Hecke character that generates this character group.

```
> _<x> := PolynomialRing(Integers());
> K := NumberField(x^4+4*x^3+7*x^2+2*x+1);
> f := Polynomial([1, 1-K.1, 1+K.1+K.1^2, 1]);
> E := ext<K|f>; // E/K is cyclic of deg 3
> G := HeckeCharacterGroup(E); psi:=G.1;
> A := AbelianExtension(psi); // either psi or G
> assert IsIsomorphic(E,NumberField(A));
> L := LSeries(psi);
> CFENew(L);
-2.3665827156630354162351856958 5E-30
```

Here is an example that is not cyclic, namely the Hilbert class field of $\mathbf{Q}(\sqrt{-4027})$.

```
> K := QuadraticField(-4027);
> ClassGroup(K);
Abelian Group isomorphic to Z/3 + Z/3
> G := HeckeCharacterGroup(1*Integers(K));
> L := AbelianExtension(G);
> assert G eq HeckeCharacterGroup(L);
> assert IsIsomorphic(HilbertClassField(K),NumberField(L));
> L1 := AbelianExtension(G.1); // L1/K cyclic deg 3
> L2 := AbelianExtension(G.2); // L2/K cyclic deg 3
> assert L1*L2 eq L;
```

41.5 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): FldAb -> RngOrdIdl, [RngIntElt]`

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): FldAb -> RngIntElt`

The absolute discriminant of A as a number field over $\mathbb{Q}$.

`Conductor(A): FldAb -> RngOrdIdl, [RngIntElt]`

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 [HPP97, Pau96].

`Degree(A): FldAb -> RngIntElt`

The degree of the abelian extension A.

`AbsoluteDegree(A): FldAb -> RngIntElt`

The degree of the abelian extension A over $\mathbb{Q}$.

`CoefficientRing(A): FldAb -> Fld`

`CoefficientField(A): FldAb -> Fld`

`BaseField(A): FldAb -> Fld`

The base field of the abelian extension A, that is `FieldOfFractions(BaseRing(A))`.

`BaseRing(A): FldAb -> Rng`

`CoefficientRing(A): FldAb -> Rng`

The base ring of the abelian extension A, that is the maximal order used to define the underlying ray class group.

`NormGroup(A): FldAb -> Map, RngOrdIdl, [RngIntElt]`

The norm group (see the definition of *norm groups*) used to define the abelian extension A.

`DecompositionField(p, A): RngOrdIdl, FldAb -> FldAb`

The decomposition field of the finite prime p in the abelian extension A as an abelian (sub)extension.

`DecompositionField(p, A): PlcNumElt, FldAb -> FldAb`

The decomposition field of the place p in the abelian extension A as an abelian extension.

`DecompositionGroup(p, A): RngIntElt, FldAb -> GrpAb`

`DecompositionGroup(p, A): RngOrdIdl, FldAb -> GrpAb`

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): PlcNumElt, FldAb -> GrpAb`

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): FldAb, RngOrdIdl -> [Tpl]`

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): FldAb, PlcNumElt -> [Tpl]`

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): FldAb, RngIntElt -> [Tpl]`

`Normal: BoolElt`	`Default: false`

The "type" of the decomposition over $\mathbb{Q}$ 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): FldAb, [ ] -> Mset`

`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): FldAb, RngIntElt, RngIntElt -> Mset`

`Normal: BoolElt`	`Default: false`

Computes the decomposition type over $\mathbb{Q}$ in the abelian extension A of all prime numbers $a \leq p \leq b$ 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.

41.6 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): FldAb -> Map

For an abelian extension A of F, this returns the Artin map as a map from the set of ideals of F to the group of automorphisms of A over F. (This induces an isomorphism from the defining group of A to the group of automorphisms).

Since this function constructs the number field defined by A, this may involve a lengthy calculation.

FrobeniusAutomorphism(A, p): FldAb, RngOrdIdl -> Map

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): FldAb -> GrpFP, [Map], Map

```
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): FldAb -> GrpFP, SeqEnum

```
Factor: RingIntElt                Default: 1
```

In case of A and it's base field k both begin normal over $\mathbb{Q}$, the automorphism group G of $A/\mathbb{Q}$ is a group extension of the abelian group coming from the definition of A and the automorphism group of $k/\mathbb{Q}$. This functions sets up the corresponding group extension problem and uses `DistinctExtensions(CM): ModCoho → SeqEnum` 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): FldAb, SeqEnum -> GrpFP, SeqEnum

```
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: Probable Automorphism Group of a Class Field (ex-b351e9)

We will demonstrate the use of `ProbableAutomorphismGroup` by investigating extensions of $\mathbb{Q}(\sqrt{10})$:

```
> 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
```

(continues on next page)

(continued from previous page)

```
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:

```
> _ := 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  [ 0.006993296366796325415873541339 07,
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): FldAb -> GrpPerm, SeqEnum, GaloisData

Given an abelian extension, compute its Galois group over $\mathbb{Q}$, ie. the abstract automorphism group of a $\mathbb{Q}$-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 a 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): FldAb -> UserProgram

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-cocycle 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*.

41.7 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): FldAb, RngOrdElt, RngOrdIdl -> BoolElt`

`IsLocalNorm(A, x, p): FldAb, RngIntElt, RngInt -> BoolElt`

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. The second signature applies when the base field is the rationals, so that x is a rational integer and p a prime ideal of the integers.

`IsLocalNorm(A, x, i): FldAb, RngOrdElt, RngIntElt -> BoolElt`

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): FldAb, RngOrdElt, PlcNumElt -> BoolElt`

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): FldAb, RngOrdElt -> BoolElt`

Returns `true` if and only if x is a local norm everywhere in the abelian extension A.

`Knot(A): FldAb -> GrpAb`

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): FldAb, RngOrdElt -> BoolElt, [RngOrdElt]`

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): FldAb, RngOrdElt -> BoolElt`

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: Norm Equations via Class Fields (ex-56eb0a)

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 $\mathbb{Q}(\zeta_5)(\eta)$ where $\eta^{20} - \zeta_5\eta^{10} + \zeta_5^2 = 0$ over the cyclotomic field. This field has degree 80 over $\mathbb{Q}$ 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 +
```

(continues on next page)

(continued from previous page)

```
        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.

41.8 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.

41.8.1 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: RngOrd -> [Rec]`

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 $\mathbb{Q}(\zeta_k)$, the degree of O over o may be smaller than expected. Furthermore, Oa will be in optimized representation.

Example: Hilbert Class Field of an Imaginary Quadratic Field (ex-44588f)

We will demonstrate this with the Hilbert class field of $\mathbb{Q}(\sqrt{-1001})$:

```
> 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
```

(continues on next page)

(continued from previous page)

```
Index: 4100090871676479535981/1
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.

41.8.2 Abelian Extensions

Abelian extensions have several attributes. Most of them are only useful in programming.

`A`Components: FldAb -> [Rec]

This read-only attribute 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 $\mathbb{Z}$. 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 `ClassField.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: FldAb -> Rec

A`NormGroup: FldAb -> Rec

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` ∘ `GrpMap`.

A`IsAbelian: FldAb -> Bool

Stores the result of a call to *IsAbelian* with parameter `All := ``true`.

A`IsNormal: FldAb -> Bool

Stores the result of a call to *IsNormal* with parameter `All := ``true`.

A`IsCentral: FldAb -> Bool

Stores the result of a call to *IsCentral* with parameter `All := ``true`.

Example: Attributes of a Ray Class Field (ex-63b1eb)

To illustrate the preceding examples we will investigate the 3-part of the ray class field modulo 36 over $\mathbb{Z}[\sqrt{-10}]$:

```
> 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)]];
> a;
-11/1*$.2 + 4/1*$.3 + 2/1*$.4
> c[1]`Gen;
[0, -11, 4, 2]
> c[1]`Gen eq a;
true
> c[1]`GenInv * a eq 1;
```

(continues on next page)

(continued from previous page)

```
true
> c[2]`GenInv * a eq 1;
false
```

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
```

41.9 Group Theoretic Functions

41.9.1 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): [] -> GrpFp, Map`

```
Mult  : Intrinsic                    Default: '*'
Eq    : Intrinsic                    Default: 'eq'
Id    : Any                          Default:
GrpGen: Verbose                      Default: 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): GrpFP, . -> BoolElt, GrpFP, Map`

```
GrpGen: Verbose                    Default: 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 [Dim] to find all elements of G with as few operations as possible.

`FindGenerators(G): GrpFP -> []`

Given a generic group G as returned by `GenericGroup`, find a small set of generators.

41.10 Bibliography

- [Dim] "Dimino's Algorithm." 13–23.
- [Coh00] Henri Cohen. *Advanced Topics in Computational Number Theory*. Springer, Berlin–Heidelberg–New York, 2000.
- [CDO] H. Cohen, F. Diaz y Diaz, M. Olivier. "Computing Ray Class Groups, Conductors and Discriminants." 52–59.
- [CDO98] Henri Cohen, Francisco Diaz y Diaz, and Michel Olivier. Computing Ray Class Groups, Conductors and Discriminants. *Math. Comp.*, 67(222):773–795, 1998.
- [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. Master's thesis, Technische Universität Berlin, 1996. 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(5):552–567, 2012. doi:10.1016/j.jsc.2011.12.053.

Chapter 42

Étale Algebras

42.1 Introduction

An *étale algebra* A over a field K is a finite product of finite separable extensions $K_1, \ldots, K_n$ of K. Typical examples are:

(a) $A = K \times K$ where K is a number field.

(b) $A = K[x]/f(x)$ where $f(x)$ is a polynomial in $K[x]$ and no repeated roots over $\overline{K}$.

We will refer to the field K as the *prime field* of A and to the fields $K_1, \ldots, K_n$ as the *components* of A. If F is a finite extension of K such that $K_1, \ldots, K_n$ are all defined as relative extensions of F, we call F the *base field* of A. If the components $K_1, \ldots, K_n$ of A have distinct defining polynomials, say $f_1(x), \ldots, f_n(x) \in F[x]$ then A is isomorphic to the étale algebra $F[x]/(f(x))$ where $f(x) = f_1(x) \cdot \cdots \cdot f_n(x)$. The polynomial $f(x)$ is then referred to as the *defining polynomial* of A.

This section describes functions to work with these algebras, orders inside the algebra, and ideals of the orders.

Currently all number fields involved must be absolute number fields.

An étale algebra has type `AlgEtQ`, an order has type `AlgEtQOrd`, an ideal has type `AlgEtQIdl`, and the elements have type `AlgEtQElt`.

This was released in version 2.29 and the original code was written by Stefano Marseglia. Furthermore, his papers, [Mar20, Mar24, Mar25], are a good reference for many of the algorithms implemented in this package.

42.2 Construction

`EtaleAlgebra(seq): SeqEnum[FldNum] -> AlgEtQ`

Given a sequence of number fields returns the étale algebra corresponding to the direct product. Note: the number fields with `DefiningPolynomial` of degree one should be created with the parameter `DoLinearExtension` set to `true`.

Example: Etale Algebra of Two Copies of Q `(ex-e44d0f)`

We now consider the étale algebra consisting of two copies of the rational field.

```
> _<x> := PolynomialRing(Integers());
> QQ := NumberField(x-1:DoLinearExtension);
> A := EtaleAlgebra([QQ,QQ]);
> a := PrimitiveElement(A); a;
<1, 2>
```

`EtaleAlgebra(f): RngUPolElt[RngInt] -> AlgEtQ`

`EtaleAlgebra(f): RngUPolElt[FldRat] -> AlgEtQ`

Given a squarefree polynomial over the integers or rationals returns the product of the number fields defined by the irreducible factors.

`DirectProduct(seq): SeqEnum[AlgEtQ] -> AlgEtQ, SeqEnum[Map], SeqEnum[Map]`

Given a sequence of étale algebras over $\mathbb{Q}$, returns their direct product, together with the natural inclusions and projections.

42.3 Attributes

`DefiningPolynomial(A): AlgEtQ -> RngUPolElt`

Returns the defining polynomial of the algebra A, if the corresponding number fields are distinct.

`Components(A): AlgEtQ -> SeqEnum`

Returns the number fields of which the algebra A is a product of, together with embeddings and projections.

`Dimension(A): AlgEtQ -> RngInt`

Returns the dimension of the algebra A.

`AbsoluteDimension(A): AlgEtQ -> RngInt`

Return the dimension of the algebra A over the prime field.

`HasBaseField(A): AlgEtQ -> BoolElt`

Returns whether the components of the algebra A have a common base field.

`BaseField(A): AlgEtQ -> FldNum, Map`

Returns the common base field of the algebra A, if it exists.

`PrimeField(A): AlgEtQ -> FldNum`

Returns the prime field of the algebra A.

`IsNumberField(A): AlgEtQ -> BoolElt, FldNum, Map`

Returns whether A has a single component (hence is a number field). If so, also returns the number field and an isomorphism from A to it.

`A1 eq A2: AlgEtQ, AlgEtQ -> BoolElt`

Returns whether the algebras $A1$ and $A2$ are equal.

42.4 Homomorphisms

`HomsToC(A): AlgEtQ -> SeqEnum[Map]`

Precision: RngIntElt	Default: Precision(GetDefaultRealField())

Returns the sequence of homomorphisms from the algebra A to a complex field $\mathbb{C}$. The precision of $\mathbb{C}$ is given by the optional parameter `Precision`. The default value is 30.

Example: Complex Embeddings of an Etale Algebra (ex-58e95c)

```
> _<x> := PolynomialRing(Integers());
> A := EtaleAlgebra(x^2+2);
> homs := HomsToC(A : Precision := 30);
> #homs;
2
```

`Hom(A, B, img): AlgEtQ, AlgEtQ, SeqEnum[AlgEtQElt] -> Map`

```
CheckMultiplicative: BoolElt                    Default: false
CheckUnital        : BoolElt                    Default: false
ComputeInverse     : BoolElt                    Default: true
```

Given étale algebras A and B and a sequence `img` of elements of B with length equal to the absolute dimension of A, returns the $\mathbb{Q}$-algebra homomorphism sending the `AbsoluteBasis(A)` to `img`. If `ComputeInverse` is true and the map is invertible, preimages are defined. If `CheckMultiplicative` (resp. `CheckUnital`) is `true`, multiplicativity (resp. unitality) is checked.

`DiagonalEmbedding(K, V): AlgEtQ, AlgEtQ -> Map`

`NaturalAction(K, V): AlgEtQ, AlgEtQ -> Map`

Let $K = K_1 \times \cdots \times K_n$ be a product of distinct number fields and $V = K_1^{s_1} \times \cdots \times K_n^{s_n}$. Returns the natural component-wise diagonal embedding $K \to V$.

Example: Homomorphism into a Product Algebra (ex-aab1bc)

```
> _<x> := PolynomialRing(Integers());
> A := EtaleAlgebra((x^2+2)*(x^2+3));
> B := EtaleAlgebra(Components(A));
> // Build a hom by mapping AbsoluteBasis(A) into B component-wise
> img := [ B!Components(AbsoluteBasis(A)[i]) : i in [1..AbsoluteDimension(A)] ];
> m := Hom(A,B,img : CheckMultiplicative := false, CheckUnital := false,
> ComputeInverse := true);
> // Diagonal embedding on a suitable power algebra
> V, embs, projs := DirectProduct([A,A]);
> d := DiagonalEmbedding(A,V);
> d(One(A));
<1, 1, 1, 1>
```

42.5 Elements

`Parent(x): AlgEtQElt -> AlgEtQ`

`Algebra(x): AlgEtQElt -> AlgEtQ`

Returns the algebra to which the element x belongs to.

`Components(x): AlgEtQElt -> SeqEnum`

Given an element x, returns its components, which are elements of number fields.

`AbsoluteCoordinates(x): AlgEtQElt -> SeqEnum`

Given an element x, returns the coordinates relative to the absolute basis, which are elements of the prime rational field.

`AbsoluteCoordinates(x, S): AlgEtQElt, AlgEtQOrd -> SeqEnum`

Given an element x and an order S, returns the coordinates of x with respect to the stored Z-basis of S.

`IsCoercible(A, x): AlgEtQ, Any -> BoolElt, AlgEtQElt`

Return whether the element x is coercible into the algebra A and the result of the coercion if so.

`A ! x: AlgEtQ, Any -> AlgEtQElt`

Coerce x into the algebra A.

`One(A): AlgEtQ -> AlgEtQElt`

The multiplicative neutral element of the algebra A.

`Zero(A): AlgEtQ -> AlgEtQElt`

The additive neutral element of the algebra A.

`IsUnit(x): AlgEtQElt -> BoolElt`

Returns whether the element x is a unit in its algebra A.

`IsZeroDivisor(x): AlgEtQElt -> BoolElt`

Returns whether the element x is a zero-divisor in its algebra A.

`Random(A, bd): AlgEtQ, RngIntElt -> AlgEtQElt`

Returns a random element of the algebra A. The coefficients are bounded by the positive integer bd.

`Random(A): AlgEtQ -> AlgEtQElt`

`bd: RngIntElt` Default: 3

Returns a random element of the algebra A. The coefficients are bounded by the parameter bd (default 3).

`RandomUnit(A, bd): AlgEtQ, RngIntElt -> AlgEtQElt`

Returns a random unit of the algebra A. The coefficients are bounded by the positive integer bd.

`x1 eq x2: AlgEtQElt, AlgEtQElt -> BoolElt`
`x1 eq x2: RngIntElt, AlgEtQElt -> BoolElt`
`x1 eq x2: AlgEtQElt, RngIntElt -> BoolElt`
`x1 eq x2: FldRatElt, AlgEtQElt -> BoolElt`
`x1 eq x2: AlgEtQElt, FldRatElt -> BoolElt`

Returns whether the elements $x1$ and $x2$ are equal.

`x1 + x2: AlgEtQElt, AlgEtQElt -> AlgEtQElt`
`x1 + x2: Any, AlgEtQElt -> AlgEtQElt`
`x1 + x2: AlgEtQElt, Any -> AlgEtQElt`
`- x: AlgEtQElt -> AlgEtQElt`
`x1 - x2: AlgEtQElt, AlgEtQElt -> AlgEtQElt`
`x1 - x2: Any, AlgEtQElt -> AlgEtQElt`
`x1 - x2: AlgEtQElt, Any -> AlgEtQElt`
`x1 * x2: AlgEtQElt, AlgEtQElt -> AlgEtQElt`
`x1 * x2: Any, AlgEtQElt -> AlgEtQElt`
`x1 * x2: AlgEtQElt, Any -> AlgEtQElt`
`Inverse(x): AlgEtQElt -> AlgEtQElt`

The multiplicative inverse of the algebra element x.

`x ^ n: AlgEtQElt, RngIntElt -> AlgEtQElt`
`x1 / x2: AlgEtQElt, AlgEtQElt -> AlgEtQElt`
`x1 / x2: Any, AlgEtQElt -> AlgEtQElt`
`x1 / x2: AlgEtQElt, Any -> AlgEtQElt`
`&+ seq: SeqEnum[AlgEtQElt] -> AlgEtQElt`

Given a sequence of `AlgEtQElt` returns the sum of the entries.

`&* seq: SeqEnum[AlgEtQElt] -> AlgEtQElt`

Given a sequence of `AlgEtQElt` returns the product of the entries.

`DotProduct(a, b): SeqEnum, SeqEnum -> Any`

Given two sequences $a = [a_1, \ldots, a_n]$ and $b = [b_1, \ldots, b_n]$, returns $\sum_i a_i \cdot b_i$.

Example: Components and Idempotents of an Etale Algebra (ex-1d7031)

```
> _<x>:=PolynomialRing(Integers());
> f := (x^8+16)*(x^8+81);
> A := EtaleAlgebra(f);
> // We compute the `canonical` primitive element, which is the class of the
> // variable x in A.
> a := PrimitiveElement(A); a;
<$.1, $.1>
> // The algebra A has two components:
> comps, embeddings, projections:=Components(A);
> K1, K2 := Explode(comps);
> // The unit element of each component corresponds to an orthogonal idempotent
> //                                                                    of A:
> [ embeddings[1](K1!1),embeddings[2](K2!1) ] eq OrthogonalIdempotents(A);
true
> // We conclude this example by showing the use of DotProduct and
> //                                              its timings advantages:
> N := 10^5;
> elts1 := [ a+i : i in [1..N] ];
> elts2 := [ a-i : i in [1..N] ];
> time s1 := &+[ elts1[i]*elts2[i] : i in [1..N] ];
Time: 0.450
> time s2 := DotProduct(elts1,elts2);
Time: 0.150
> s1 eq s2;
true
```

`MinimalPolynomial(x): AlgEtQElt -> RngUPolElt`

Returns the minimal polynomial over the common base ring of the number fields defining the algebra A of the element x.

`MinimalPolynomial(x, F): AlgEtQElt, Rng -> RngUPolElt`

Returns the minimal polynomial over the ring F of the element x.

`AbsoluteMinimalPolynomial(x): AlgEtQElt -> RngUPolElt`

Returns the minimal polynomial over the prime field of the element x or an algebra.

`IsIntegral(x): AlgEtQElt -> BoolElt`

Returns whether the element x of an algebra is integral (over the integers).

`Evaluate(f, a): RngUPolElt, AlgEtQElt -> AlgEtQElt`

Evaluate the polynomial f at the algebra element a.

`PrimitiveElement(A): AlgEtQ -> AlgEtQElt`

Returns the primitive element of the étale algebra A. Note that A has a primitive element only if it is the product of distinct number fields.

Given an étale algebra A over $\mathbb{Q}$ there exists an element $a \in A$ such that $A = \mathbb{Q}[a]$, that is, every element can be written as a polynomial with rational coefficients in a. Such an element is called a *primitive element* of A. It is characterized by having a minimal polynomial whose degree equals the absolute dimension of A.

The intrinsic `PrimitiveElement` produces such an element of the étale algebra A using a deterministic procedure which we now describe: Let N be the number of components of A, each one having primitive element a_i. Set $b_1 = a_1$. For $i = 2, \ldots, N$, set $b_i = a_i + j$ where j is the smallest non-negative integer such that the minimal polynomial of $a_i + j$ is not in the set of minimal polynomials of the elements $b_1, \ldots, b_{i-1}$. The output is the element of A whose components are $b_1, ..., b_N$. In particular, if A is a product of number fields with different defining polynomials, then the output is the element of A whose components are the primitive elements of the components.

`PowerBasis(A): AlgEtQ -> SeqEnum[AlgEtQElt]`

Returns the power basis of the étale algebra A, consisting of powers of the primitive element of A.

`Basis(A): AlgEtQ -> SeqEnum`

Returns a basis of the algebra A over the common base field.

`AbsoluteBasis(A): AlgEtQ -> SeqEnum`

Returns a basis of the algebra A over the prime field.

`A . i: AlgEtQ, RngIntElt -> AlgEtQElt`

Returns the i-th element of the absolute basis of A.

`AbsoluteCoordinates(seq, basis): SeqEnum[AlgEtQElt], SeqEnum[AlgEtQElt] -> SeqEnum`

Given a sequence of elements and a basis over the `PrimeField` returns a sequence whose entries are the coordinates in the `PrimeField` with respect to the given basis.

`OrthogonalIdempotents(A): AlgEtQ -> SeqEnum`

Returns the orthogonal idempotent element of the étale algebra A.

`Idempotents(A): AlgEtQ -> SeqEnum`

Returns the idempotent element of the étale algebra A.

42.6 Orders of Algebras

`IsCoercible(S, x): AlgEtQOrd, Any -> BoolElt, AlgEtQElt`

Return whether x is coercible into the order S and the result if so.

`Order(gens): SeqEnum[AlgEtQElt] -> AlgEtQOrd`

```
Check            : RngIntElt                    Default: 100
CheckIsKnownOrder: BoolElt                      Default: true
```

Construct the order generated by *gens* over the rationals. The parameter `Check` (default 100) determines how many times the program tries to obtain a multiplicatively closed lattice by adding the product of the generators. If `Check` is 0 then this step is skipped. The parameter `CheckIsKnownOrder` determines whether the program checks if the order is already known, i.e. in the attribute `KnownOrders` of the algebra. This is to avoid the creation of multiple copies of the same order. The default value is `true`.

`Order(A, orders): AlgEtQ, Tup -> AlgEtQOrd`

Given a sequence of orders in the number fields defining the étale algebra A, generates the direct sum order.

`Algebra(S): AlgEtQOrd -> AlgEtQ`

Returns the algebra of the order S.

`ZBasis(S): AlgEtQOrd -> SeqEnum[AlgEtQElt]`

Return a Z-basis of the order S.

`Generators(S): AlgEtQOrd -> SeqEnum[AlgEtQElt]`

Return a set of generators (as a Z-algebra) of the order S.

`O1 eq O2: AlgEtQOrd, AlgEtQOrd -> BoolElt`

Checks equality of orders in an étale algebra.

Equality of orders is performed using the `Hash` attribute which is constructed as follows. Let S be an order. Let P be the upper triangular Hermite normal form of the integer square matrix $d \cdot M$ where M is the matrix whose rows are the coefficients of a $\mathbb{Z}$-basis of S and d is the least common denominator of its entries. The `Hash` of S is defined to be the sequence consisting of the least common denominator of $\frac{1}{d} \cdot P$ and the entries of the upper triangular part of $\frac{1}{d} \cdot P$. This hashing

method has no collisions and it is independent of the choice of $\mathbb{Z}$-basis from which we start the procedure.

`x in O: AlgEtQElt, AlgEtQOrd -> BoolElt`

`x in O: RngIntElt, AlgEtQOrd -> BoolElt`

`x in O: FldRatElt, AlgEtQOrd -> BoolElt`

Return whether the element x is contained in the order O of an algebra.

`AbsoluteCoordinates(seq, O): SeqEnum[AlgEtQElt], AlgEtQOrd -> SeqEnum`

Returns the coordinates of the elements in `seq` with respect to the stored Z-basis of O.

`One(S): AlgEtQOrd -> AlgEtQElt`

Returns the unit element of the order S.

`Zero(S): AlgEtQOrd -> AlgEtQElt`

Returns the zero element of the order S.

`Random(O, bd): AlgEtQOrd, RngIntElt -> AlgEtQElt`

```
ZeroDivisorsAllowed: BoolElt                    Default: false
```

Returns a random element of the order O. The coefficients are bounded by the positive integer bd. One can allow zero-divisors using the optional parameter `ZeroDivisorsAllowed`, which is set to `false` by default.

`Random(O): AlgEtQOrd -> AlgEtQElt`

```
CoeffRange          : RngIntElt                 Default: 3
ZeroDivisorsAllowed: BoolElt                    Default: false
```

Returns a random (small coefficient) element of the order O. The range of the random coefficients can be increased by giving the optional parameter `CoeffRange`. One can allow zero-divisors using the optional parameter `ZeroDivisorsAllowed`, which is set to `false` by default.

`IsKnownOrder(~R): AlgEtQOrd`

This procedure checks whether the order R is already in the list of known orders of the algebra A containing R. If so then it replaces R with the copy stored in the attribute `KnownOrders`. If not it adds it to `KnownOrders`. This is done to avoid creating multiple copies of the same order.

`EquationOrder(A): AlgEtQ -> AlgEtQOrd`

Given an étale algebra defined by a polynomial, returns the monogenic order defined by the same polynomial.

`ProductOfEquationOrders(A): AlgEtQ -> AlgEtQOrd`

Given an étale algebra A, returns the order consisting of the product of the equation orders of the number fields.

`MaximalOrder(A): AlgEtQ -> AlgEtQOrd`

Returns the maximal order of the étale algebra A. It is the direct sum of the ring of integers of the number fields composing the algebra.

`IsMaximal(S): AlgEtQOrd -> BoolElt`

Returns whether the order S is the maximal order of the étale algebra.

`IsProductOfOrders(O): AlgEtQOrd -> BoolElt, Tup`

Return if the order O is a product of orders in number fields, and if so return also the sequence of these orders.

`IsProductOfOrdersInComponents(O): AlgEtQOrd -> BoolElt, Tup`

Returns whether the argument is a product of orders in the components of its parent algebra. If so, it returns also a tuple containing these orders.

`IsProductOfOrdersInFactorAlgebras(S): AlgEtQOrd -> BoolElt, SeqEnum[AlgEtQElt]`

Returns whether the given order is a product of orders living in some factor algebras of the parent algebra. This is equivalent to containing some idempotents of the algebra other than 0 and 1. If this is the case, it returns also the idempotents.

Example: Orders in a Product of Number Fields (ex-f008b6)

```
> _<x> := PolynomialRing(Integers());
> // We consider the following three number fields
> K1 := NumberField(x^2-2);
> K2 := NumberField(x^2-3);
> K3 := NumberField(x^2-5);
> // We define the product \'etale algebra A and the factor algebras B and C
> // consisting of only the first two components and the last one, respectively.
> B := EtaleAlgebra([K1,K2]);
> C := EtaleAlgebra([K3]);
> A, embs, projs := DirectProduct([B,C]);
> // The maximal order of A is the product of the three ring of integers.
> OA := MaximalOrder(A);
> IsProductOfOrdersInComponents(OA);
true <Maximal Equation Order with defining polynomial x^2 - 2 over its ground
order, Maximal Equation Order with defining polynomial x^2 - 3 over its ground
order, Maximal Order of Equation Order with defining polynomial x^2 - 5 over
its ground order>
> // The equation order of A is not a product in any factor algebra
> EA := EquationOrder(A);
> IsProductOfOrdersInFactorAlgebras(EA);
false []
> // Now we construct an order that is a product of an order in B and one in C,
> // but does not admit further splittings.
> a := PrimitiveElement(A);
> e1 := A![1,0,0];
> e2 := A![0,1,1];
> R := Order([a*e1,a*e2]);
> IsProductOfOrdersInFactorAlgebras(R);
true [ <1, 0, 0>, <0, 1, 1> ]
```

`Index(T): AlgEtQOrd -> FldRatElt`

Given an order T compute its index with respect to the basis of the algebra of T as a free $\mathbb{Z}$-module.

`Index(S, T): AlgEtQOrd, AlgEtQOrd -> FldRatElt`

Given two orders $T \subset S$, returns $[S : T] = \#S/T$.

`O1 subset O2: AlgEtQOrd, AlgEtQOrd -> BoolElt`

Checks whether $O1$ is contained in $O2$.

`O1 * O2: AlgEtQOrd, AlgEtQOrd -> AlgEtQOrd`

Returns the order generated by the orders $O1$ and $O2$.

`O1 meet O2: AlgEtQOrd, AlgEtQOrd -> AlgEtQOrd`

Return the intersection of orders $O1$ and $O2$.

`MultiplicatorRing(R): AlgEtQOrd -> AlgEtQOrd`

Returns the multiplicator ring of an order R, that is R itself.

42.7 Ideals

`Ideal(S, gens): AlgEtQOrd, SeqEnum -> AlgEtQIdl`

Creates an ideal of the order S, generated by the elements of *gens*.

`Ideal(S, idls): AlgEtQOrd, Tup -> AlgEtQIdl`

Given an order S which is a product of orders S_i in the number fields generating `Algebra(S)`, and a tuple of ideals I_i of S_i, returns the S-ideal corresponding to the direct sum of the I_i.

`Ideal(S, gen): AlgEtQOrd, Any -> AlgEtQIdl`

`S * gen: AlgEtQOrd, AlgEtQElt -> AlgEtQIdl`

`S * gen: AlgEtQOrd, RngIntElt -> AlgEtQIdl`

`S * gen: AlgEtQOrd, FldRatElt -> AlgEtQIdl`

`gen * S: AlgEtQElt, AlgEtQOrd -> AlgEtQIdl`

`gen * S: RngIntElt, AlgEtQOrd -> AlgEtQIdl`

`gen * S: FldRatElt, AlgEtQOrd -> AlgEtQIdl`

Creates an ideal of S, generated by *gen*.

`T !! I: AlgEtQOrd, AlgEtQIdl -> AlgEtQIdl`

Given an S-ideal I and an order T, returns the extension IT as a T-ideal. Note that if T is a subset of S, then $IT = I$.

`Algebra(I): AlgEtQIdl -> AlgEtQ`

Returns the étale algebra in which the ideal I lives.

`Order(I): AlgEtQIdl -> AlgEtQOrd`

Returns the order of definition of the ideal I.

`ZBasis(I): AlgEtQIdl -> SeqEnum[AlgEtQElt]`

Returns a $\mathbb{Z}$-basis of the ideal I.

`Generators(I): AlgEtQIdl -> SeqEnum[AlgEtQElt]`

Returns the generators of the ideal I.

`I eq J: AlgEtQIdl, AlgEtQIdl -> BoolElt`

`I ne J: AlgEtQIdl, AlgEtQIdl -> BoolElt`

Returns whether the ideals I and J are equal, respectively not equal.

Let I a fractional S-ideal in an étale algebra over $\mathbb{Q}$. Let P be the upper triangular Hermite normal form of the integer square matrix $d \cdot M$ where M is the matrix whose rows are the coefficients of a $\mathbb{Z}$-basis of I and d is the least common denominator of its entries. The `Hash` of I is defined to be the sequence consisting of the least common denominator of $\frac{1}{d} \cdot P$ and the entries of the upper triangular part of $\frac{1}{d} \cdot P$. This hashing method has no collisions and it is independent of the choice of $\mathbb{Z}$-basis from which we start the procedure.

`I eq S: AlgEtQIdl, AlgEtQOrd -> BoolElt`

`S eq I: AlgEtQOrd, AlgEtQIdl -> BoolElt`

Return whether I is equal to S when I is an ideal of S.

`AbsoluteCoordinates(x, I): AlgEtQElt, AlgEtQIdl -> SeqEnum`

Given an element x and an ideal I, returns the coordinates of x with respect to the stored Z-basis of I.

`AbsoluteCoordinates(seq, I): SeqEnum[AlgEtQElt], AlgEtQIdl -> SeqEnum`

Returns the coordinates of the elements in `seq` with respect to the stored Z-basis of I.

`x in I: AlgEtQElt, AlgEtQIdl -> BoolElt`

`x in I: RngIntElt, AlgEtQIdl -> BoolElt`

`x in I: FldRatElt, AlgEtQIdl -> BoolElt`
Returns whether the element x is in the ideal I.

`S subset I: AlgEtQOrd, AlgEtQIdl -> BoolElt`
Given an ideal I of an order S, return whether $S \subseteq I$.

`I subset S: AlgEtQIdl, AlgEtQOrd -> BoolElt`
Given an ideal I of an order S, return whether $I \subseteq S$.

`I1 subset I2: AlgEtQIdl, AlgEtQIdl -> BoolElt`
Checks if the ideal $I1$ is inside the ideal $I2$. The ideals need to be fractional.

`Index(T): AlgEtQIdl -> FldRatElt`
Given an ideal T computes its index with respect to the basis of the algebra of T as a free $\mathbb{Q}$-module.

`Index(J, I): AlgEtQIdl, AlgEtQIdl -> Any`
Given fractional ideals J and I defined over the same order returns $[J : I] = [J : J \cap I]/[I : J \cap I]$.

`Index(S, I): AlgEtQOrd, AlgEtQIdl -> Any`
Given an ideal I of an order S returns $[S : I] = [S : S \cap I]/[I : S \cap I]$.

`OneIdeal(S): AlgEtQOrd -> AlgEtQIdl`
Given an order S returns the ideal $1 * S$ which will be cached.

`Conductor(O): AlgEtQOrd -> AlgEtQOrdIdl`
Computes the conductor of an order O, defined as the colon ideal $(O : O_K)$, where O_K is the maximal order of the algebra.

`I + J: AlgEtQIdl, AlgEtQIdl -> AlgEtQIdl`
Returns the sum of two ideals.

`I * J: AlgEtQIdl, AlgEtQIdl -> AlgEtQIdl`
Product of two ideals.

`I * x: AlgEtQIdl, AlgEtQElt -> AlgEtQIdl`
`I * x: AlgEtQIdl, RngIntElt -> AlgEtQIdl`
`I * x: AlgEtQIdl, FldRatElt -> AlgEtQIdl`
`x * I: AlgEtQElt, AlgEtQIdl -> AlgEtQIdl`
`x * I: RngIntElt, AlgEtQIdl -> AlgEtQIdl`
`x * I: FldRatElt, AlgEtQIdl -> AlgEtQIdl`
Returns $x * I$.

`I ^ n: AlgEtQIdl, RngIntElt -> AlgEtQIdl`
Returns the nth power of an ideal.

`I meet S: AlgEtQIdl, AlgEtQOrd -> AlgEtQIdl`
`S meet I: AlgEtQOrd, AlgEtQIdl -> AlgEtQIdl`
Given an ideal I of S, return $S \cap I$.

`I meet J: AlgEtQIdl, AlgEtQIdl -> AlgEtQIdl`
Given ideals I and J, return $J \cap I$.

`&+ seq: SeqEnum[AlgEtQIdl] -> AlgEtQIdl`
Returns the sum of the fractional ideals in the sequence.

`ColonIdeal(I, J): AlgEtQIdl, AlgEtQIdl -> AlgEtQIdl`
Computes the colon ideal $(I : J)$ (as an O-ideal) of two O-ideals, which is the set of elements x of the algebra such that $x \cdot J \subset I$.

`ColonIdeal(O, J): AlgEtQOrd, AlgEtQIdl -> AlgEtQIdl`
Computes the colon ideal $(1 \cdot O : J)$ (as an O-ideal).

`ColonIdeal(I, O): AlgEtQIdl, AlgEtQOrd -> AlgEtQIdl`

Computes the colon ideal $(I : 1 \cdot O)$ (as an O-ideal).

`IsInvertible(I): AlgEtQIdl -> BoolElt`

Checks if the ideal I is invertible in its order of definition O.

`Inverse(I): AlgEtQIdl -> AlgEtQIdl`

Computes the inverse of an invertible ideal I.

`MultiplicatorRing(I): AlgEtQIdl -> AlgEtQOrd`

Given a fractional ideal I computes its multiplicator ring $(I : I)$.

`IsProductOfIdeals(I): AlgEtQIdl -> BoolElt, Tup`

Return if the ideal I is a product of ideals in the number fields defining the algebra. If so, it returns also the sequence of these ideals (in the appropriate orders). Note: we require `Order(I)` to be `MultiplicatorRing(I)`.

`Random(I, bd): AlgEtQIdl, RngIntElt -> AlgEtQElt`

```
ZeroDivisorsAllowed: BoolElt                    Default: false
```

Returns a random element of the ideal I. The coefficients are bounded by the positive integer *bd*. One can allow zero-divisors using the optional parameter `ZeroDivisorsAllowed`, which by default is set to `false`.

`Random(I): AlgEtQIdl -> AlgEtQElt`

```
CoeffRange         : RngIntElt                  Default: 3
ZeroDivisorsAllowed: BoolElt                    Default: false
```

Returns a random (small coefficient) element of the ideal I. The range of the random coefficients can be increased by giving the optional parameter `CoeffRange`. One can allow zero-divisors using the optional parameter `ZeroDivisorsAllowed`, which by default is set to `false`.

`IsCoprime(I, J): AlgEtQIdl, AlgEtQIdl -> BoolElt`

Given two integral ideals I and J of an order S, returns whether $I + J = R$.

`IsIntegral(I): AlgEtQIdl -> BoolElt`

Returns whether the ideal I of S is integral, that is $I \subseteq S$.

`MakeIntegral(I): AlgEtQIdl -> AlgEtQIdl, RngIntElt`

Given a fractional S ideal I, returns the ideal $d \cdot I, d$ when d is the smallest integer such that $d \cdot I$ is integral in S. Compare with *SmallRepresentative*.

`MinimalInteger(I): AlgEtQIdl -> RngIntElt`

Returns the smallest integer contained in the ideal I.

`CoprimeRepresentative(I, J): AlgEtQIdl, AlgEtQIdl -> AlgEtQElt, AlgEtQIdl`

Returns an element x such that $x \cdot I$ is an integral ideal coprime with J, together with the product $x \cdot I$. The first ideal must be invertible and the second should be integral.

`ZBasisLLL(~S): AlgEtQOrd`

`ZBasisLLL(~S): AlgEtQIdl`

A procedure that replaces the ZBasis with an LLL-reduced one. Note: the attribute inclusion matrix, which depends on the $\mathbb{Z}$-Basis is modified as well.

42.8 Quotients

`Quotient(I, zbJ): AlgEtQIdl, SeqEnum[AlgEtQElt] -> GrpAb, Map`

Given an ideal I and the ZBasis of an ideal or order J such that $J \subset I$, returns the abelian group $Q = I/J$ together with the quotient map $q : I \mapsto Q$.

`Quotient(I, J): AlgEtQIdl, AlgEtQIdl -> GrpAb, Map`

Given fractional ideals $J \subset I$, returns the abelian group $Q = I/J$ together with the quotient map $q : I \mapsto Q$.

`Quotient(S, zbJ): AlgEtQOrd, SeqEnum[AlgEtQElt] -> GrpAb, Map`

Given an order S and the ZBasis of an ideal J such that $J \subset S$, returns the abelian group $Q = S/J$ together with the quotient map $q : S \mapsto Q$. The ideal J can also be an order.

`ResidueRing(S, I): AlgEtQOrd, AlgEtQIdl -> GrpAb, Map`

Given an integral ideal I of S, returns the abelian group S/I and the quotient map $q : S \mapsto S/I$ (with preimages). Important: the domain of q is the Algebra of S, since the elements of S are expressed as elements of A. We stress that the output is a group and does not have a multiplication. This can be obtained by first taking preimages, doing the multiplication, and then applying the projection.

`ResidueField(P): AlgEtQIdl -> FldFin, Map`

Given P a prime of S, returns a finite field F isomorphic to S/P and a surjection (with inverse) $S \mapsto F$.

`PrimitiveElementResidueField(P): AlgEtQIdl -> AlgEtQElt`

Returns an element of the ideal P that maps to the primitive element of the residue field S/P, that is a multiplicative generator of $(S/P)^*$.

`QuotientVS(I, J, P): AlgEtQOrd, AlgEtQOrd, AlgEtQIdl -> ModRng, Map`

Let I, J be orders, P a fractional R-ideal such that:

- P is prime of some order R with residue field K;
- J in I and I/J is a vector space V over K, say of dimension d.

The function returns the `KModule` $K^d = V$ and the natural surjection $I \mapsto V$ (with preimages).

Example: Residue Field and Residue Ring (ex-c0d808)

```
> _<x> := PolynomialRing(Integers());
> f := (x^8+16)*(x^8+81);
> A := EtaleAlgebra(f);
> O := MaximalOrder(A);
> // Choose a prime above 2
> P := PrimesAbove(2*O)[1];
> // Residue field
> F, pi := ResidueField(P);
> F;
Finite field of size 2
> // Residue ring
> G, q := ResidueRing(O,P);
> G;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
2*G.1 = 0
```

`QuotientVS(I, J, P): AlgEtQOrd, AlgEtQIdl, AlgEtQIdl -> ModRng, Map`

Let I be an order, J and P be fractional R-ideals such that:

- P is prime of some order R, with residue field K;
- J in I and I/J is a vector space V over K, say of dimension d.

The function returns the `KModule` $K^d = V$ and the natural surjection $I \mapsto V$ (with preimages).

`QuotientVS(I, J, P): AlgEtQIdl, AlgEtQOrd, AlgEtQIdl -> ModRng, Map`

Let J be an order, I and P be fractional R-ideals such that:

- P is prime of some order R, with residue field K;
- J in I and I/J is a vector space V over K, say of dimension d.

The function returns the `KModule` $K^d = V$ and the natural surjection $I \mapsto V$ (with preimages).

`QuotientVS(I, J, P): AlgEtQIdl, AlgEtQIdl, AlgEtQIdl -> ModRng, Map`

Let I, J, P be fractional R-ideals such that:

- P is prime of some order R;
- J in I and I/J is a vector space over R/P, say of dimension d;

The function returns the `KModule` $K^d = V$ and the natural surjection $I \mapsto V$ (with preimages).

42.9 Over Orders

`IsMaximalAtPrime(R, P): AlgEtQOrd, AlgEtQIdl -> BoolElt`

Returns whether R is maximal at the prime P, that is, if $(R : O)$ is not contained in P, where O is the maximal order.

`MinimalOverOrdersAtPrime(R, P): AlgEtQOrd, AlgEtQIdl -> SetIndx[AlgEtQOrd]`

Given an order R and prime P of R, it returns the minimal overorders S of R with conductor $(R : S)$ which is P-primary. The minimality assumption forces the conductor $(R : S)$ to be exactly P. Based on [HS20].

`MinimalOverOrders(R): AlgEtQOrd -> SetIndx[AlgEtQOrd]`

Computes the minimal overorders of R.

`OverOrdersAtPrime(R, P): AlgEtQOrd, AlgEtQIdl -> SeqEnum[AlgEtQOrd]`

Given an order R and prime P of R, it returns R and the overorders S of R with conductor $(R : S)$ which is P-primary. We recursively produce the minimal PP-overorders where PP are primes above P. Based on [HS20].

`OverOrders(R): AlgEtQOrd -> SeqEnum[AlgEtQOrd]`

```
populateoo_in_oo: BoolElt                    Default: false
```

We compute all the overorders of R. The parameter `populateoo_in_oo` (default `false`) determines whether we should fill the attribute `T`OverOrders` for every overorder T of R. The computation is based on [HS20].

Example: Over-Orders of an Equation Order `(ex-1d4978)`

```
> _<x> := PolynomialRing(Integers());
> f := (x^4+16)*(x^4+81);
> A := EtaleAlgebra(f);
> E := EquationOrder(A);
> oo := OverOrders(E);
> #oo;
2288
> pp := SingularPrimes(E);
> // We see that the size of the lattice of inclusions of the overorders is the
> // product of the sizes of local component of the lattice.
> #oo eq &*[ #OverOrdersAtPrime(E,P) : P in pp ];
true
```

(continues on next page)

(continued from previous page)

```
> // Now we consider only the P-overorders S for the first singular prime P.
> // We verify that there is always a positive integer i such that (R:S)^i is
> // invertible in its multiplicator ring.
> ooP := OverOrdersAtPrime(E,pp[1]);
> #ooP;
11
> forall{exists{IsInvertible(Ti!!Ci) where Ti:=MultiplicatorRing(Ci)
> where Ci:=C^i : i in [1..10]} where C := ColonIdeal(E,E!!OneIdeal(S))
>                                                          : S in ooP};
true
```

`FindOverOrders(R): AlgEtQOrd -> SetIndx[AlgEtQOrd]`

```
populateoo_in_oo: BoolElt                    Default: false
```

We compute all the overorders of R. The parameter `populateoo_in_oo` (default `false`) determines whether we should fill the attribute `T`OverOrders` for every overorder T of R. The computation is based on [HS20].

42.10 Over Order Graph

`GraphOverOrders(R): AlgEtQOrd -> GrphDir`

Given an order R returns the graph G of minimal inclusions of the overorders of R. More precisely, the vertices of G are integers between 1 and the number of `OverOrders(R)`, and there is an edge $[i, j]$ if and only if `OverOrder(R)[j]` is a minimal overorder of `OverOrders(R)[i]`.

42.11 Trace and Norm

Let A be an étale algebra over $\mathbb{Q}$, with components $K_1 \times \cdots \times K_n$. We define the *(absolute) trace* on A as the additive map $\mathrm{Tr}_{A/\mathbb{Q}} \colon A \to \mathbb{Q}$ that sends an element $a \in A$ to $\sum_{i=1}^{n} \mathrm{Tr}_{K_i/\mathbb{Q}}(a)$. Let m_a be the matrix representing the multiplication-by-a on A with respect to any basis of A over $\mathbb{Q}$. Then $\mathrm{Tr}_{A/\mathbb{Q}}(a)$ equals the trace of m_a.

We define the *(absolute) norm* on A as the multiplicative map $\mathrm{N}_{A/\mathbb{Q}} \colon A \to \mathbb{Q}$ by sending a unit $a \in A$ to $\prod_{i=1}^{n} \mathrm{N}_{K_i/\mathbb{Q}}(a)$ and every zero-divisor to 0. We have $N_{A/\mathbb{Q}}(a)$ equals the determinant of the matrix m_a.

`Trace(x): AlgEtQElt -> FldRatElt`

Returns the trace of the element x of an étale algebra.

`Norm(x): AlgEtQElt -> FldRatElt`

Returns the norm of the element x of an étale algebra.

`AbsoluteTrace(x): AlgEtQElt -> FldRatElt`

Returns the absolute trace of the element x of an étale algebra. Since the étale algebra is over the rationals this is the same as `Trace`.

`AbsoluteNorm(x): AlgEtQElt -> FldRatElt`

Returns the absolute norm of the element x of an étale algebra. Since the étale algebra is over the rationals this is the same as `Norm`.

`TraceDualIdeal(I): AlgEtQIdl -> AlgEtQIdl`

Returns the trace dual ideal of the ideal I, that is, the set of elements x of the algebra such that $\mathrm{Tr}(x \cdot I)$ is integer-valued.

Let I be an order or a fractional ideal in an étale algebra A over $\mathbb{Q}$. We defined the *trace dual ideal* of I as $I^t = \{a \in A : \mathrm{Tr}_{A/\mathbb{Q}}(a \cdot I) \subseteq \mathbb{Z}\}$. For fractional ideals I and J and a unit $a \in A$, we have:

(a) if $I \subseteq J$ then $J^t \subseteq I^t$ and $\#(J/I) = \#(I^t/J^t)$;

(b) $(aI)^t = \frac{1}{a} I^t$;

(c) $(I + J)^t = I^t \cap J^t$;

(d) $(I \cap J)^t = I^t + J^t$;

(e) $(I : J)^t = I^t \cdot J$.

`TraceDualIdeal(O): AlgEtQOrd -> AlgEtQIdl`

Returns the trace dual ideal of an order in an étale algebra, that is, the set of elements x of the algebra such that $\mathrm{Tr}(x \cdot O)$ is integer-valued.

42.12 Completion

`Completion(P): AlgEtQIdl -> FldPad, Map`

```
MinPrecision: RngIntElt                    Default: 20
```

Given a prime ideal P of the maximal order of an étale algebra L, returns the p-adic field corresponding to the completion L_P and a homomorphism $\phi : L \to L_P$ with preimages. The parameter `MinPrecision` is passed to `Completion`.

Example: Uniformizers and Completions (ex-ecd376)

```
> _<x> := PolynomialRing(Integers());
> f := (x^8+16)*(x^8+81);
> A := EtaleAlgebra(f);
> O := MaximalOrder(A);
> // Consider a bunch of prime of O and their uniformizers in O.
> pp := PrimesAbove(2*3*5*7*O);
> unifs := Uniformizers(pp);
> // We now verify that each element is a uniformizer at the correct prime
> // and a unit everywhere else
> assert Matrix([[ Valuation(mP(t)) : t in unifs ] where AP, mP := Completion(P)
>                                                    : P in pp]) eq␣
↪1;
```

42.13 Intermediate Ideals

`IntermediateIdeals(I, J): AlgEtQIdl, AlgEtQIdl -> SetIndx[AlgEtQIdl]`

```
Minimal                    : BoolElt                    Default: false
Maximal                    : BoolElt                    Default: false
PrescribedMultiplicatorRing: BoolElt                    Default: false
```

Given fractional S-ideals $J \subset I$, returns all the fractional S-ideals K such that $J \subset K \subset I$.

If `Minimal` is set true, only the minimal ideals are returned. If `Maximal` is set true, only the maximal ideals are returned. If `PrescribedMultiplicatorRing` is set true, only ideals K with $(K : K) = S$ are returned. The computation is done recursively starting with the minimal or maximal ones.

`IntermediateIdeals(I, J, O): AlgEtQIdl, AlgEtQIdl, AlgEtQOrd -> SetIndx[AlgEtQIdl]`

```
PrescribedMultiplicatorRing: BoolElt                    Default: false
```

Given fractional S-ideals I and J and an order O such that $S \subseteq O$, $J \subseteq I$, and $O \subseteq (I : I)$, this function returns all the fractional S-ideals K such that

- $J \subseteq K \subseteq I$, and
- $O \cdot K = I$.

If `PrescribedMultiplicatorRing` is set true, then the output contains only K such that $(K : K) = S$. Note that the output may contain I. The output is produced by recursively computing maximal intermediate ideals.

`IntermediateIdeals(I, J, N): AlgEtQIdl, AlgEtQIdl, RngIntElt -> SetIndx[AlgEtQIdl]`

Given ideals $J \subset I$ over the same order, and a positive integer N, it returns all the ideals K such that

- $J \subset K \subset I$, and
- $[I : K] = N$.

These are computed by recursively searching for maximal submodules.

42.14 Ideals of Index

`IdealsOfIndex(O, N): RngOrd, RngIntElt -> SeqEnum[RngOrdIdl]`

Given an order O in a number field and a positive integer N, returns all the ideals I of index $[O : I] = N$.

`IdealsOfIndex(I, N): RngOrdIdl, RngIntElt -> SeqEnum[RngOrdIdl]`

Given an ideal I in an order O in a number field and a positive integer N, with N coprime with the conductor, returns all the ideals J contained in I with index $[I : J] = N$.

`IdealsOfIndex(I, N): RngOrdFracIdl, RngIntElt -> SeqEnum[RngOrdFracIdl]`

Given an ideal I in an order O in a number field and a positive integer N, with N coprime with the conductor, returns all the ideals J contained in I with index $[I : J] = N$.

`IdealsOfIndex(I, N): AlgEtQIdl, RngIntElt -> SeqEnum[AlgEtQIdl]`

```
Method: MonStgElt                    Default: "Default"
```

Given an O-ideal I in O and a positive integer N, returns all the subideals J of I with index $[I : J] = N$. The function is very fast if N is coprime to the conductor of O. If this condition is not satisfied a slow algorithm is used which doesn't require additional hypothesis. One can force the slow algorithm by setting the parameter `Method:="Slow"`.

`IdealsOfIndex(O, N): AlgEtQOrd, RngIntElt -> SeqEnum[AlgEtQIdl]`

```
Method: MonStgElt                    Default: "Default"
```

Given an order O and a positive integer N, returns all the O-ideals J with index $[O : J] = N$. The function is very fast if N is coprime to the conductor of O. If this condition is not satisfied a slow algorithm is used which doesn't require additional hypothesis. One can force the slow-naive algorithm by setting the parameter `Method:="Slow"`.

42.15 Short Element and Small Representative

`ShortElement(I): AlgEtQIdl -> AlgEtQElt`

Given an ideal I returns a non-zerodivisor in I with small coefficients (in the LLL sense). This is achieved by randomly picking an element with small coefficients in a LLL-reduced basis (wrt the T2 norm as a $\mathbb{Z}$-lattice).

`SmallRepresentative(I): AlgEtQIdl -> AlgEtQIdl, AlgEtQElt`

Given a fractional R-ideal I, it returns an isomorphic ideal $a \cdot I$, and the element a, such that $a \cdot I$ is a subset of R, and the cardinality of R/aI is small. This is achieved by computing the `ShortElement` a of $(R : I)$. Note that if I is invertible R/aI is isomorphic to $(R : I)/aR$.

42.16 Minimal Generators

`TwoGeneratingSet(I): AlgEtQIdl`

A procedure that given an invertible ideal I put in the attribute `Generators` of I two non-zerodivisors in I that generate I. If I is known to be principal nothing is done.

42.17 Chinese Remainder Theorem

Let I and J be integral fractional ideals over the same order S in an étale algebra. Assume that I and J are coprime, that is, $I + J = S$. Then $I \cap J = I \cdot J$ and we have a canonical S-linear isomorphism

$$\frac{S}{I \cap J} \simeq \frac{S}{I} \times \frac{S}{J}.$$

`ChineseRemainderTheorem(Is, as): SeqEnum[AlgEtQIdl], SeqEnum[AlgEtQElt] -> AlgEtQElt`

Given a sequence Is of ideals of S, pairwise coprime, and a sequence as of elements of S, it returns an element e such that $e - as[i] \in Is[i]$ for every i.

`ChineseRemainderTheorem(I, J, a, b): AlgEtQIdl, AlgEtQIdl, AlgEtQElt, AlgEtQElt -> AlgEtQElt`

Given two coprime ideals I and J of S, two elements $a, b \in S$, finds e such that $(e - a) \in I$ and $(e - b) \in J$.

`ChineseRemainderTheoremFunctions(Is): SeqEnum[AlgEtQIdl] -> Map, Map`

Given a sequence Is of N integral fractional S-ideals $I_1, \ldots, I_N$, pairwise coprime, returns a map $S \to S^N$ representing the natural isomorphism $\frac{S}{I} \to \frac{S}{I_1} \times \cdots \times \frac{S}{I_N}$, where $I = \prod_i I_i$, and a map $S^N \to S$ representing the inverse.

Example: Chinese Remainder Theorem Maps (ex-d7dd94)

```
> _<x> := PolynomialRing(Integers());
> A := EtaleAlgebra((x^2+2)*(x^2+3));
> O := MaximalOrder(A);
> I1 := PrimesAbove(2*O)[1];
> I2 := PrimesAbove(3*O)[1];
> toProd, fromProd := ChineseRemainderTheoremFunctions([I1,I2]);
> toProd(One(A));
[
<1, 0>,
<1, 0>
]
```

42.18 Picard Group

Let R be an order in an étale algebra A over $\mathbb{Q}$ with maximal order $\mathcal{O}_A$. We say that two fractional R-ideals I and J are *isomorphic* if there exists a unit $a \in A$ such that $I = aJ$. Observe that this happens if and only if I and J are isomorphic as R-modules. We refer to the isomorphism class $[I]$ of I as its *ideal class*.

The set of ideal classes of invertible fractional R-ideals forms a group under the operation induced by ideal multiplication. This group is called the *Picard group* of R and denoted $\mathrm{Pic}(R)$.

If $K_1 \times \ldots \times K_n$ are the components of A then $\mathrm{Pic}(\mathcal{O}_A) = \prod_{i=1}^{n} \mathrm{Cl}(\mathcal{O}_{K_i})$, where $\mathrm{Cl}(\mathcal{O}_{K_i})$ is the class group of the number field K_i. Also, the unit group $\mathcal{O}_A^\times$ of $\mathcal{O}_A$ satisfies $\mathcal{O}_A^\times = \mathcal{O}_{K_1}^\times \times \ldots \times \mathcal{O}_{K_n}^\times$.

The Picard group and the unit group of the order R can be computed using the well-known exact sequence:

$$1 \to R^\times \to \mathcal{O}_A^\times \to \frac{(\mathcal{O}_A/\mathfrak{f})^\times}{(R/\mathfrak{f})^\times} \to \mathrm{Pic}(R) \to \mathrm{Pic}(\mathcal{O}_A) \to 1,$$

where the first, second, and third map are the natural maps, the fourth is induced by $a \mapsto (a\mathcal{O}_A \cap R)$ and the last one is induced by the extension map $I \mapsto I\mathcal{O}_A$.

`ResidueRingUnits(S, I): AlgEtQOrd, AlgEtQIdl -> GrpAb, Map`

Returns the group $(S/I)^*$ and a map $(S/I)^* \to S$. The order S is required to be maximal.

`ResidueRingUnits(I): AlgEtQIdl -> GrpAb, Map`

Given a fractional S-ideal I, returns the group $(S/I)^*$ and a map $(S/I)^* \to S$ giving representatives. Implemented when S is maximal.

`ResidueRingUnitsSubgroupGenerators(F): AlgEtQIdl -> SeqEnum[AlgEtQElt]`

Given a fractional S-ideal F, returns generators of $(S/F)^*$.

`IsPrincipal(I1): AlgEtQIdl -> BoolElt, AlgAssElt`

GRH: BoolElt	Default: false

Return if the argument is a principal ideal; if so the function returns also the generator. The optional parameter `GRH` decides whether the bound for the `IsPrincipal` test should be conditional. The default value is `false`.

`PicardGroup(S): AlgEtQOrd -> GrpAb, Map`

GRH: BoolElt	Default: false

Return the `PicardGroup` of the order S, which is not required to be maximal, and a map from the Picard group to a set of representatives of the ideal classes. The optional parameter `GRH` decides the bound for the computations of the Class group and Unit group of the maximal order. The default value is `false`.

`ExtensionHomPicardGroups(S, T): AlgEtQOrd, AlgEtQOrd -> Map`

GRH: BoolElt	Default: false

Given orders S and T such that $S \subseteq T$, returns the surjective extension map $\mathrm{Pic}(S) \to \mathrm{Pic}(T)$. The parameter `GRH` is passed to `PicardGroup`.

`UnitGroup(S): AlgEtQOrd -> GrpAb, Map`

GRH: BoolElt	Default: false

Return the unit group of a order in a étale algebra. The optional parameter `GRH` decides the bound for the computation of the unit group of the maximal order. The default value is `false`.

Example: Picard Group and Units of an Order (ex-034565)

```
> _<x> := PolynomialRing(Integers());
> A := EtaleAlgebra((x^4+16)*(x^4+81));
> E := EquationOrder(A);
> P, phi := PicardGroup(E);
> AbelianInvariants(P);
[ 2, 24, 24, 24, 24 ]
```

(continues on next page)

(continued from previous page)

```
> U, psi := UnitGroup(E);
> TorsionInvariants(U);
[ 2 ]
> TorsionFreeRank(U);
2
```

`IsIsomorphic(I, J): AlgEtQIdl, AlgEtQIdl -> BoolElt, AlgAssElt`

`GRH: BoolElt` Default: `false`

Checks if $I = x \cdot J$, for some x. If so, also x is returned. The optional parameter `GRH` decides whether the bound for the `IsPrincipal` test should be conditional. The default value is `false`.

42.19 Factorization and Primes

`Factorization(I): AlgEtQIdl -> Tup`

Given an integral S-ideal I coprime with the conductor of S (hence invertible in S), returns its factorization into a product of primes of S.

`PrimesAbove(I): AlgEtQIdl -> SeqEnum[AlgAssEtOrdIdl]`

Given an integral S-ideal I, returns the sequence of maximal ideals P of S above I.

`SingularPrimes(R): AlgEtQOrd -> SeqEnum[AlgAssEtOrdIdl]`

Returns the non-invertible primes of the order R.

`PlacesAboveRationalPrime(E, p): AlgEtQ, RngIntElt -> SeqEnum[AlgEtQIdl]`

Given an étale algebra and a rational prime, returns the primes of the maximal order of the algebra containing the rational prime.

`NonInvertiblePrimes(R): AlgEtQOrd -> SetIndx`

Returns the non-invertible primes of the order R.

`IsPrime(I): AlgEtQIdl -> BoolElt`

`IsMaximal(I): AlgEtQIdl -> BoolElt`

`IsMaximalIdeal(I): AlgEtQIdl -> BoolElt`

Given an integral S-ideal I, returns if the ideal is a prime fractional ideal of S, that is a maximal S ideal.

`Valuation(x, P): AlgEtQElt, AlgEtQIdl -> RngIntElt`

`Valuation(I, P): AlgEtQIdl, AlgEtQIdl -> RngIntElt`

Valuation at the prime P of an element x or of a fractional ideal I of the maximal order.

`InertiaDegree(P): AlgEtQIdl -> RngIntElt`

`RamificationIndex(P): AlgEtQIdl -> RngIntElt`

For a prime P of the maximal order $\mathcal{O}$, returns its inertia degree and ramification index.

`IsBassAtPrime(S, P): AlgEtQOrd, AlgEtQIdl -> BoolElt`

Check if the order is Bass at the prime ideal P, that is, if every overorder of S is Gorenstein at the primes above P.

`IsBass(S): AlgEtQOrd -> BoolElt`

Check if the order S is Bass, that is, if every overorder of S is Gorenstein.

`IsGorensteinAtPrime(S, P): AlgEtQOrd, AlgEtQIdl -> BoolElt`

Check if the order S is Gorenstein at the prime ideal P, that is, if every fractional ideal I with $(I : I) = S$ is locally principal at P.

`IsGorenstein(O): AlgEtQOrd -> BoolElt`

Checks if the order O is Gorenstein, that is if the `TraceDualIdeal` of O is invertible, or equivalently, if all fractional ideals I with $(I : I) = O$ are invertible.

`Uniformizers(PPs): SeqEnum[AlgEtQIdl] -> SeqEnum`

Given a sequence of primes P of the maximal order, returns a sequence of elements t_P such that t_P is a uniformizer at P and a unit at every other prime in the sequence.

42.20 Low Cohen Macauley Type

`NonGorensteinPrimes(S): AlgEtQOrd -> SeqEnum, SeqEnum`

Given an order S it returns two sequences: the first containis the primes at which S is locally not Gorenstein; the second contains the Cohen Macaulay types of S at these primes, that is, the dimension of S^t/PS^t over S/P, where S^t is the `TraceDualIdeal` of S.

`CohenMacaulayTypeAtPrime(S, P): AlgEtQOrd, AlgEtQIdl -> RngIntElt`

Given an order S and a prime ideal P, it returns its Cohen-Macaulay Type at P. This integer equals the dimension of $S^t/P * S^t$ where S^t is the trace dual of S.

`CohenMacaulayType(S): AlgEtQOrd -> RngIntElt`

Given an order S returns its Cohen-Macaulay Type. This integer equals the max dimension of $S^t/P * S^t$ where S^t is the trace dual of S and P runs over all (non-Gorenstein) primes of S.

Example: Order of Maximal Cohen-Macaulay Type (ex-442922)

```
> // Given an \'etale algebra A of dimension n over Q, the maximum Cohen-Macaulay
> // type that an order can have is n-1.
> // An order with such maximal Cohen-Macauulay type can be found among the
> // overorders of the order Z+p*O_A where p an arbitrary rational prime.
> // We verify this statement in an example.
> _<x> := PolynomialRing(Integers());
> f := x^4+8;
> A := EtaleAlgebra(f);
> O := MaximalOrder(A);
> p := 7;
> E := Order([p*z : z in ZBasis(O)]);
> n := Dimension(A);
> oo := OverOrders(E);
> exists{ S : S in oo | CohenMacaulayType(S) eq n-1 };
true
```

42.21 Weak Classes

Let R be an order and I and J be fractional R-ideals in an étale algebra A over $\mathbb{Q}$. Then the following statements are equivalent:

(a) $I_\mathfrak{p} \simeq J_\mathfrak{p}$ as $R_\mathfrak{p}$-modules for every maximal ideal $\mathfrak{p}$ of R.

(b) $I_p \simeq J_p$ as R_p-modules for every rational prime p.

(c) $1 \in (I : J)(J : I)$.

(d) I and J have the same multiplicator ring, say S, and $(I : J)$ is an invertible fractional S-ideal.

(e) I and J have the same multiplicator ring, say S, and $(J : I)$ is an invertible fractional S-ideal. If the previous conditions hold, one says that I and J are *weakly equivalent*. This is an equivalence relation.

Note that if I is an order, then J is weakly equivalent to I if and only if J is an invertible I-ideal. If both I and J are orders then they are weakly equivalent if and only if they are equal. In the literature, the same notion sometimes goes under the name *local isomorphism*.

`WKICM_bar(S): AlgEtQOrd -> SeqEnum`

`Method: MonStgElt` `Default: "Auto"`

Returns representatives of all the weak equivalence classes I of S satisfying $(I : I) = S$. The parameter `Method` (default `"Auto"`) determines if we should use the `"IntermediateIdeals"` routine or the `"LowIndexProcess"`, which is potentially much slower but more memory efficient.

`WeakEquivalenceClassesWithPrescribedMultiplicatorRing(S): AlgEtQOrd -> SeqEnum[AlgEtQIdl]`

Given an order S, returns representatives I of all weak equivalence classes of fractional S-ideals with $(I : I) = S$.

We also provide an abstract representation of the weak equivalence class monoid. In this representation the monoid has type `AlgEtQWECM` and classes have type `AlgEtQWECMElt`. Classes can be created via the coercion operator `!`, multiplied with `*` (and exponentiated with `^`), and a representative can be recovered using the `RepresentativeMap`. Additional helpers include `IsOne`, `IsIdempotent`, and `Localization`.

Let R be an order in an étale algebra over $\mathbb{Q}$. Then ideal multiplication induces a commutative monoid structure on the set of weak equivalence classes of fractional R-ideals called the *weak equivalence class monoid* of R. This commutative monoid, which we denote by $\mathcal{W}(R)$, is finite and has unit element given by the class of R.

Since the multiplicator ring is an invariant of the weak equivalence class of a fractional R-ideal, we have a partitioning $\mathcal{W}(R) = \bigsqcup_S \mathcal{W}_S(R)$, where the disjoint union is taken over the overorders S of R and $\mathcal{W}_S(R)$ is the subset of $\mathcal{W}(R)$ consisting of the weak equivalence classes of fractional R-ideals with multiplicator ring S.

Let S be an overorder of R. If S is Gorenstein then $\mathcal{W}_S(R)$ consists only of the class defined by S. If S has Cohen-Macaulay type 2 and $\mathfrak{p}$ is a prime of S, then each fractional R-ideal I with multiplicator ring S satisfies either $I_\mathfrak{p} \simeq S_\mathfrak{p}$ or $I_\mathfrak{p} \simeq S_\mathfrak{p}^t$. The second possibility occurs precisely when S has Cohen-Macaulay type 2 at the primes $\mathfrak{p}$. This classification results provide a very efficient method to compute $\mathcal{W}_S(R)$ for orders S with Cohen-Macaulay type ≤ 2. For the general case, an algorithm that includes a more expensive enumeration step is provided.

`WeakEquivalenceClassMonoid(E): AlgEtQOrd -> SeqEnum[AlgEtQIdl]`

`WKICM(E): AlgEtQOrd -> SeqEnum`

`Method: MonStgElt` `Default: "Auto"`

Computes the weak equivalence class monoid of E. The parameter `Method` (default `"Auto"`) determines if we should use the `"IntermediateIdeals"` routine or the `"LowIndexProcess"`, which is potentially much slower but more memory efficient.

The second method to compute $\mathcal{W}(R)$ returns an abstract representation of the monoid together with a map giving representatives. The monoid is of type `AlgEtQWECM` and each class is of type `AlgEtQWECMElt`. Classes can be created by feeding to the operator `!` a fractional S-ideal, when S is an overorder of R, or an overorder of R itself. Whenever a class is created a representative is stored in an attribute. This representative can be changed using the intrinsic `SetRepresentative`. Classes can be multiplied using the operator `*`. Whenever two classes are multiplied, the result is stored in the attribute `MultiplicationTable` of the monoid.

`WeakEquivalenceClassMonoidAbstract(R): AlgEtQOrd -> AlgEtQWECM, Map`

Given an order R, returns the abstract weak equivalence class monoid W of R together with a map (with preimages) sending each class to a representative.

`W ! x: AlgEtQWECM, Any -> AlgEtQWECMElt`

Coerce x into the monoid W, when possible, returning a class.

`x in W: AlgEtQWECMElt, AlgEtQWECM -> BoolElt`

Returns whether the class x belongs to the monoid W.

`Parent(x): AlgEtQWECMElt -> AlgEtQWECM`
`Ideal(x): AlgEtQWECMElt -> AlgEtQIdl`
`MultiplicatorRing(x): AlgEtQWECMElt -> AlgEtQOrd`
`x1 eq x2: AlgEtQWECMElt, AlgEtQWECMElt -> BoolElt`
`SetRepresentative(x, J): AlgEtQWECMElt, AlgEtQIdl`

Change the stored representative of the class x to the weakly equivalent ideal J.

`MultiplicationTable(W): AlgEtQWECM -> Assoc`
`x * y: AlgEtQWECMElt, AlgEtQWECMElt -> AlgEtQWECMElt`
`x ^ n: AlgEtQWECMElt, RngIntElt -> AlgEtQWECMElt`
`IsOne(x): AlgEtQWECMElt -> BoolElt`
`IsIdempotent(x): AlgEtQWECMElt -> BoolElt`
`Order(W): AlgEtQWECM -> AlgEtQOrd`
`Array(W): AlgEtQWECM -> Assoc`
`RepresentativeMap(W): AlgEtQWECM -> Map`
`W1 eq W2: AlgEtQWECM, AlgEtQWECM -> BoolElt`
`# W: AlgEtQWECM -> RngInt`
`Classes(W): AlgEtQWECM -> SeqEnum[AlgEtQWECMElt]`
`Representatives(W): AlgEtQWECM -> SeqEnum[AlgEtQIdl]`
`One(W): AlgEtQWECM -> AlgEtQWECMElt`
`Random(W): AlgEtQWECM -> AlgEtQWECMElt`
`Idempotents(W): AlgEtQWECM -> SeqEnum[AlgEtQWECMElt]`
`Localization(W, P): AlgEtQWECM, AlgEtQIdl -> AlgEtQWECM`
`IsGeneratingSet(W, seq): AlgEtQWECM, SeqEnum[AlgEtQWECMElt] -> BoolElt`
`IsGeneratingSet(W, seq): AlgEtQWECM, SeqEnum[AlgEtQIdl] -> BoolElt`

(More standard intrinsics.)

Example: Weak Equivalence Classes (ex-628357)

```
> _<x> := PolynomialRing(Integers());
> f := (x^4+16)*(x^4+81);
> A := EtaleAlgebra(f);
> E := EquationOrder(A);
> w := WKICM_bar(E);
> #w;
1
> // Optionally compute full monoid
> W := WKICM(E);
> #W;
```

42.22 Weak Testing

`IsWeakEquivalent(I, J): AlgEtQIdl, AlgEtQIdl -> BoolElt`
`IsWeaklyEquivalent(I, J): AlgEtQIdl, AlgEtQIdl -> BoolElt`

Checks if I and J are weakly equivalent, that is, if $1 \in (I : J)(J : I)$, or equivalently, if I and J are locally equivalent at all prime of their common multiplicator ring. This function does not require that the ideals are defined over the same order.

`IsWeakEquivalent(O1, O2): AlgEtQOrd, AlgEtQOrd -> BoolElt`

`IsWeaklyEquivalent(O1, O2): AlgEtQOrd, AlgEtQOrd -> BoolElt`

Check if the two orders $O1$ and $O2$ are weakly equivalent, that is equal.

`IsWeakEquivalent(O, J): AlgEtQOrd, AlgEtQIdl -> BoolElt`

`IsWeaklyEquivalent(J, O): AlgEtQIdl, AlgEtQOrd -> BoolElt`

Checks if the ideal J is weakly equivalent to order O, that is, if J is invertible in O.

Example: Testing Weak Equivalence (ex-6405b0)

```
> _<x> := PolynomialRing(Integers());
> f := (x^4+16)*(x^4+81);
> A := EtaleAlgebra(f);
> E := EquationOrder(A);
> I := OneIdeal(E);
> J := I;
> IsWeakEquivalent(I,J);
true
```

`IsWeakEquivalent(J, O): AlgEtQIdl, AlgEtQOrd -> BoolElt`

Checks if the ideal J is weakly equivalent to order O, that is, if J is invertible in O.

42.23 Ideal Class Monoid

Let R be an order in an étale algebra A over $\mathbb{Q}$. Ideal multiplication induces the structure of commutative monoid on the set of ideal classes of R, which we then call *ideal class monoid* of R. We denote it by $\mathcal{I}(R)$. The unit element of $\mathcal{I}(R)$ is the class of any principal fractional R-ideal.

We have a partitioning $\mathcal{I}(R) = \bigsqcup_S \mathcal{I}_S(R)$ where the disjoint union is taken over the overorders S of R and $\mathcal{I}_S(R)$ is the subset of $\mathcal{I}(R)$ consisting of ideal classes with multiplicator ring S. The computation is then performed by first computing $\mathcal{W}(R)$ and then observing that for each overorder S of R, the Picard group $\mathrm{Pic}(S)$ acts freely on $\mathcal{I}_S(R)$ with quotient space $\mathcal{W}_S(R)$.

`ICM_bar(S): AlgEtQOrd -> SeqEnum`

`GRH: BoolElt` Default: `false`

Returns the ideal classes of fractional S-ideals having Multiplicator Ring equal to S. This is the same as the orbit of the action of `PicardGroup(S)` on `WKICM_bar(S)`.

`ICM(S): AlgEtQOrd -> SeqEnum`

`GRH: BoolElt` Default: `false`

Returns the ideal class monoid of the order S, that is, a set of representatives for the isomorphism classes of the fractional S-ideals.

We also provide an abstract representation of the ideal class monoid. The abstract monoid has type `AlgEtQICM` and classes have type `AlgEtQICMElt`. An ideal class can be created via `!` starting from an overorder or a fractional ideal, and a (deterministic) representative can be recovered using `RepresentativeMap`. Each class is internally a pair consisting of a weak equivalence class and an element of the abstract representation of $\mathrm{Pic}(S)$.

The second method to compute the ideal class monoid of an order R, returns an abstract representation of $\mathcal{I}(R)$ with type `AlgEtQICM` together with a map to a set of representatives. Each class has type `AlgEtQICMElt`, and it is internally represented as a pair consisting of a weak equivalence class (of type `AlgEtQWECMElt`) and an element of the representation of $\mathrm{Pic}(S)$ (as an abstract abelian group), where S is the corresponding multiplicator ring. This representation is more efficient than the previous one, since it does not need to compute in advance and store a representative for each

ideal class. Ideal classes can be created using the coercion operator ! starting from an overorder S of R or a fractional S-ideal. Ideal classes can be multiplied using the operator *.

IdealClassMonoidAbstract(R): AlgEtQOrd -> AlgEtQICM, Map

Given an order R, returns the abstract ideal class monoid icm together with a map (with preimages) sending each class to a representative.

icm ! x: AlgEtQICM, Any -> AlgEtQICMElt

Coerce x into the abstract ideal class monoid when possible, returning an ideal class.

x in icm: AlgEtQICMElt, AlgEtQICM -> BoolElt

Parent(x): AlgEtQICMElt -> AlgEtQICM

WEClass(x): AlgEtQICMElt -> AlgEtQWECMElt

PicClass(x): AlgEtQICMElt -> GrpAbElt, Map

Ideal(x): AlgEtQICMElt -> AlgEtQIdl

MultiplicatorRing(x): AlgEtQICMElt -> AlgEtQOrd

x1 eq x2: AlgEtQICMElt, AlgEtQICMElt -> BoolElt

x * y: AlgEtQICMElt, AlgEtQICMElt -> AlgEtQICMElt

x ^ n: AlgEtQICMElt, RngIntElt -> AlgEtQICMElt

IsOne(x): AlgEtQICMElt -> BoolElt

IsInvertibleInMultiplicatorRing(x): AlgEtQICMElt -> BoolElt

Order(icm): AlgEtQICM -> AlgEtQOrd

RepresentativeMap(icm): AlgEtQICM -> Map

icm1 eq icm2: AlgEtQICM, AlgEtQICM -> BoolElt

icm: AlgEtQICM -> RngInt

Classes(icm): AlgEtQICM -> SeqEnum[AlgEtQICMElt]

Representatives(icm): AlgEtQICM -> SeqEnum[AlgEtQIdl]

One(icm): AlgEtQICM -> AlgEtQICMElt

Random(icm): AlgEtQICM -> AlgEtQICMElt

(More standard intrinsics.)

42.24 Complex Conjugation

Let A be an étale algebra over $\mathbb{Q}$ with components $K_1 \times \cdots \times K_n$. We say that A is a *CM-étale algebra* if every component K_i is a CM-field, that is, K_i has an involution that acts as applying complex conjugation after applying any homomorphism to the complex numbers. If A is a CM-étale algebra, then it has an involution with the same property. For this reason, we call this involution *complex conjugation* and denote it as $\bar{\cdot}$.

Given an element of A, an order, or a fractional ideal in A, we say that it is *conjugate stable* if it equals its complex conjugate. An element x of A is called *totally real* if $x = \overline{x}$ and *totally imaginary* if $x = -\overline{x}$. A totally real element a is called *totally positive* (resp. *totally negative*) if $\varphi(a) > 0$ (resp. $\varphi(a) < 0$) for every homomorphism $\varphi : A \to \mathbb{C}$.

HasComplexConjugate(A): AlgEtQ -> BoolElt

Returns if the algebra A is the product of CM fields.

ComplexConjugate(x): AlgEtQElt -> AlgEtQElt

If the algebra A of the element x is a product of CM fields, it returns the complex conjugate of the argument.

IsConjugateStable(O): AlgEtQOrd -> BoolElt, AlgEtQOrd

Given an order O in a CM-étale algebra, it returns whether O is conjugate stable and the complex conjugate.

`ComplexConjugate(O): AlgEtQOrd -> AlgEtQOrd`

Given an order O in a CM-étale algebra, it returns the complex conjugate of O.

`IsConjugateStable(I): AlgEtQIdl -> BoolElt, AlgEtQIdl`

Given a fractional ideal I in a CM-étale algebra, it returns whether I is conjugate stable and the complex conjugate. Note: if the order of I is not conjugate stable, then the second output will be defined over the complex conjugate of the order.

`ComplexConjugate(I): AlgEtQIdl -> AlgEtQIdl`

If A is a product of CM fields, it returns the complex conjugate of the fractional ideal I. Note: if the order of I is not conjugate stable, then the output will be defined over the complex conjugate of the order.

42.25 Complex Multiplication

Let A be a CM-étale algebra of dimension $2g$ over $\mathbb{Q}$. Then complex conjugation acts on $\mathrm{Homs}(A, \mathbb{C})$. We denote this action with $\bar{\cdot}$. A *CM-type* Φ of A is a set of g elements of $\mathrm{Homs}(A, \mathbb{C})$ such that $\mathrm{Homs}(A, \mathbb{C}) = \Phi \sqcup \overline{\Phi}$.

Given a CM-type Φ and a totally imaginary element $b \in A^\times$, we say that b is Φ-*positive* if $\mathrm{Im}(\varphi(b)) > 0$ for every $\varphi \in \Phi$. Two totally imaginary elements b and b' in $A^\times$ represent the same CM-type Φ if and only if b/b' is totally real and totally positive.

In Magma a CM-type of a CM-algebra A has type `AlgEtQCMType` and it is determined by g homomorphisms to $\mathbb{C}$ or by a Φ-positive element.

`CMType(seq): SeqEnum[Map] -> AlgEtQCMType`

Given a sequence *seq* of homomorphisms from a CM-algebra to CC, one per conjugate pair, it returns the corresponding CMType.

Example: CM Type of an Etale Algebra `(ex-d5afe8)`

```
> _<x> := PolynomialRing(Integers());
> A := EtaleAlgebra(x^2+1);
> homs := HomsToC(A);
> PHI := CMType([homs[1]]);
> H := Homs(PHI : Precision := 30);
> #H;
1
```

`CreateCMType(seq): SeqEnum[Map] -> AlgEtQCMType`

Given a sequence *seq* of homomorphisms from a CM-algebra to CC, one per conjugate pair, it returns the corresponding CMType.

`CMType(b): AlgEtQElt -> AlgEtQCMType`

Given a totally imaginary element b, it returns the CMType PHI for which b is PHI-positive, that is, $\mathrm{Im}(\phi(b)) > 0$ for every ϕ in PHI.

`CreateCMType(b): AlgEtQElt -> AlgEtQCMType`

Given a totally imaginary element b, it returns the CMType PHI for which b is PHI-positive.

`CMPositiveElement(PHI): AlgEtQCMType -> AlgEtQElt`

Given a CMType PHI returns a totally imaginary PHI-positive element (which uniquely determines PHI).

`CMPosElt(PHI): AlgEtQCMType -> AlgEtQElt`

Given a CMType PHI returns a totally imaginary PHI-positive element (which uniquely determines PHI).

`Homs(PHI): AlgEtQCMType -> SeqEnum[Map]`

```
prec: RngIntElt                    Default: 30
```

Given an `AlgEtQCMType` PHI returns the sequence of maps to the complex field. The parameter `prec` (default value 30) determines the precision of the codomains of the maps.

`PHI1 eq PHI2: AlgEtQCMType, AlgEtQCMType -> BoolElt`

```
prec: RngIntElt                    Default: 30
```

Returns whether two cm types are equal. This happens if and only if the quotient of (any) two CMPositiveElements is totally real and totally positive.

`Precision(PHI): AlgEtQCMType -> RngIntElt`

Returns the precision of the given CM-type, that is, the codomain of each homomorphism will be `ComplexField(Precision)`.

`ChangePrecision(PHI0, prec): AlgEtQCMType, RngIntElt -> AlgEtQCMType`

Changes the precision of the given CM-type, that is, the codomain of each homomorphism will be `ComplexField(Precision)`.

`ChangePrecision(~PHI, prec): AlgEtQCMType, RngIntElt`

Changes the precision of the given CM-type, that is, the codomain of each homomorphism will be `ComplexField(Precision)`.

`AllCMTypes(A): AlgEtQ -> SeqEnum[AlgEtQCMType]`

```
Precision: RngIntElt               Default: Precision(GetDefaultRealField())
```

Returns all the `AlgEtQCMTypes` of A. The parameter `Precision` determined the precision of the codomain of the maps defining the CMTypes.

42.26 Totally Real and Positive

`IsTotallyReal(a): AlgEtQElt -> BoolElt`

Returns whether element a is totally real.

`IsTotallyRealPositive(a): AlgEtQElt -> BoolElt`

Returns whether the element a is totally positive, that is, totally real and with positive image in CC.

`TotallyRealSubAlgebra(K): AlgEtQ -> AlgEtQ, Map`

Given a CM algebra K, returns the unique totally real subalgebra, with an embedding.

`TotallyRealUnitGroup(S): AlgEtQOrd -> Grp`

Given an order S in a CM étale algebra A returns the groups of totally real units of S, as a subgroup of S^*.

`TotallyRealPositiveUnitGroup(S): AlgEtQOrd -> Grp`

Given an order S in a CM étale algebra. Returns the groups of totally positive units of S, as a subgroup of S^*.

42.27 Printing and Saving

`PrintSeqAlgEtQElt(seq): SeqEnum[AlgEtQElt] -> SeqEnum, MonStgElt`

Given a sequence of elements of an étale algebra, returns a sequence of tuples of rational numbers that can be coerced into the original algebra to obtain the input sequence. As a second output, returns a string suitable for printing to file.

`PrintWKICM(R): AlgEtQOrd -> MonStgElt`

Given an order R, returns a string encoding the weak equivalence classes of R, sorted by multiplicator ring. In particular, the overorders of R can be recovered from this string. Such a string can be printed to file; use `LoadWKICM` to load it back.

`LoadWKICM(str): MonStgElt -> AlgEtQOrd`

Given a string produced with `PrintWKICM`, returns the corresponding order R with populated attributes for overorders and weak equivalence classes.

42.28 Bibliography

- [HS20] Tommy Hofmann and Carlo Sircana. On the computation of overorders. *International Journal of Number Theory*, 16(04):857–879, 2020. URL: https://doi.org/10.1142/S179304212050044X, arXiv:https://doi.org/10.1142/S179304212050044X, doi:10.1142/S179304212050044X.
- [Mar20] Stefano Marseglia. Computing the ideal class monoid of an order. *J. Lond. Math. Soc. (2)*, 101(3):984–1007, 2020. URL: https://doi.org/10.1112/jlms.12294, doi:10.1112/jlms.12294.
- [Mar24] Stefano Marseglia. Cohen-Macaulay type of orders, generators and ideal classes. *J. Algebra*, 658:247–276, 2024. URL: https://doi.org/10.1016/j.jalgebra.2024.05.051, doi:10.1016/j.jalgebra.2024.05.051.
- [Mar25] Stefano Marseglia. Local isomorphism classes of fractional ideals of orders in étale algebras. *J. Algebra*, 673:77–102, 2025. URL: https://doi.org/10.1016/j.jalgebra.2025.02.030, doi:10.1016/j.jalgebra.2025.02.030.

Chapter 43

Dirichlet and Hecke Characters

43.1 Introduction

This chapter presents the facilities provided in Magma for Dirichlet and Hecke characters, currently implemented over number fields. For Dirichlet characters over the rationals (as `FldRat`), see Section Dirichlet Characters.

The principal constructors are `DirichletGroup` whose elements are functions on number field elements, and `HeckeCharacterGroup` whose elements are functions 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 (see Chapter *Class Field Theory*).

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.

43.1.1 Creation Functions

`DirichletGroup(I): RngOrdIdl -> GrpDrchNF`

`DirichletGroup(I, oo): RngOrdIdl, SeqEnum -> GrpDrchNF`

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): RngOrdIdl -> GrpHecke`

`HeckeCharacterGroup(I, oo): RngOrdIdl, SeqEnum -> GrpHecke`

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): GrpDrchNF -> GrpDrchNF`

Given a group of Dirichlet characters, return the subgroup that is trivial on the image of the field units in the residue ring.

`TotallyUnitTrivialSubgroup(G): GrpDrchNF -> GrpDrchNF`

Given a group of Dirichlet characters, return the subgroup that is totally trivial on the image of the field units in the residue ring. That is, it is trivial at each place individually.

`HilbertCharacterSubgroup(G): GrpHecke -> GrpHecke`

Given a group of Hecke characters, return the subgroup corresponding to the class group (those with trivial conductor).

43.1.2 Functions on Groups and Group Elements

`Modulus(G): GrpDrchNF -> RngOrdIdl, SeqEnum`

`Modulus(G): GrpHecke -> RngOrdIdl, SeqEnum`

`Modulus(chi): GrpDrchNFElt -> RngOrdIdl, SeqEnum`

`Modulus(chi): GrpHeckeElt -> RngOrdIdl, SeqEnum`

Returns the modulus ideal and a (possibly empty) sequence of real places.

`Order(chi): GrpDrchNFElt -> RngIntElt`

`Order(psi): GrpHeckeElt -> RngIntElt`

Returns the order of a Dirichlet or Hecke character.

`Random(G): GrpDrchNF -> GrpDrchNFElt`

`Random(G): GrpHecke -> GrpHeckeElt`

Returns a random element of a Dirichlet or Hecke group.

`Domain(G): GrpDrchNF -> FldNum`

`Domain(G): GrpDrchNFElt -> FldNum`

Returns the number field that is the domain for the Dirichlet character.

`Domain(G): GrpHecke -> PowIdl`

`Domain(G): GrpHeckeElt -> PowIdl`

Returns the set of ideals that is the domain for the Hecke character.

`Decomposition(chi): GrpDrchNFElt -> List`

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.

`Components(chi): GrpDrchNFElt -> Assoc`

`Components(psi): GrpHeckeElt -> Assoc`

Given a Hecke or Dirichlet character, return its components as an associative array of Dirichlet characters indexed by bad places. For a Hecke character, this is the decomposition of its `DirichletRestriction`. Note that the character need not be primitive.

`Component(chi, P): GrpDrchNFElt, RngOrdIdl -> GrpDrchNFEl`

`Component(psi, P): GrpHeckeElt, RngOrdIdl -> GrpDrchNFElt`

`Component(chi, oo): GrpDrchNFElt, RngIntElt -> GrpDrchNFEl`

`Component(psi, oo): GrpHeckeElt, RngIntElt -> GrpDrchNFElt`

`Component(chi, P): GrpDrchNFElt, PlcNumElt -> GrpDrchNFEl`

`Component(psi, P): GrpHeckeElt, PlcNumElt -> GrpDrchNFElt`

Given a Dirichlet or Hecke character and a prime ideal or place (possibly specified by an integer for an infinite place), return the corresponding Dirichlet character component. For a place that it is not ramified, the trivial character of the integer ring is returned.

`Conductor(chi): GrpDrchNFElt -> RngOrdIdl, SeqEnum`

`Conductor(psi): GrpHeckeElt -> RngOrdIdl, SeqEnum`

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): GrpDrchNFElt -> GrpDrchNFElt`

`AssociatedPrimitiveCharacter(psi): GrpHeckeElt -> GrpHeckeElt`

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): GrpDrchNFElt, GrpDrchNF -> GrpDrchNFElt`

`Restrict(psi, H): GrpHeckeElt, GrpHecke -> GrpHeckeElt`

`Restrict(chi, I): GrpDrchNFElt, RngOrdIdl -> GrpDrchNFElt`

`Restrict(psi, I): GrpHeckeElt, RngOrdIdl -> GrpHeckeElt`

`Restrict(chi, I, oo): GrpDrchNFElt, RngOrdIdl, SeqEnum -> GrpDrchNFElt`

`Restrict(psi, I, oo): GrpHeckeElt, RngOrdIdl, SeqEnum -> GrpHeckeElt`

`Restrict(G, D): GrpDrchNF, GrpDrchNF -> GrpDrchNF`

`Restrict(G, H): GrpHecke, GrpHecke -> GrpHecke`

`Restrict(G, I): GrpDrchNF, RngOrdIdl -> GrpDrchNF`

`Restrict(G, I): GrpHecke, RngOrdIdl -> GrpHecke`

`Restrict(G, I, oo): GrpDrchNF, RngOrdIdl, SeqEnum -> GrpDrchNF`

`Restrict(G, I, oo): GrpHecke, RngOrdIdl, SeqEnum -> GrpHecke`

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): GrpDrchNF, FldCyc -> GrpDrchNF`

`TargetRestriction(H, C): GrpHecke, FldCyc -> GrpDrchNF`

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): GrpDrchNFElt, RngElt`

`SetTargetRing(~ psi, e): GrpHeckeElt, RngElt`

Given a Dirichlet or Hecke character and a suitable root of unity, modify the character to take values according to this root of unity. The ring element must be a root of unity, and its order must be a multiple of the order of the character. Writing $m = \mathrm{ord}(\chi)$, if the character previously had $\chi(u) = \zeta_m^v$, it will now have $\chi(u) = (e^q)^v$ where q is $\mathrm{ord}(e)/m$.

`Extend(chi, D): GrpDrchNFElt, GrpDrchNF -> GrpDrchNFElt`

`Extend(psi, H): GrpHeckeElt, GrpHecke -> GrpHeckeElt`

`Extend(chi, I): GrpDrchNFElt, RngOrdIdl -> GrpDrchNFElt`

`Extend(psi, I): GrpHeckeElt, RngOrdIdl -> GrpHeckeElt`

`Extend(chi, I, oo): GrpDrchNFElt, RngOrdIdl, SeqEnum -> GrpDrchNFElt`

`Extend(psi, I, oo): GrpHeckeElt, RngOrdIdl, SeqEnum -> GrpHeckeElt`

`Extend(G, D): GrpDrchNF, GrpDrchNF -> GrpDrchNF`

`Extend(G, H): GrpHecke, GrpHecke -> GrpHecke`

`Extend(G, I): GrpDrchNF, RngOrdIdl -> GrpDrchNF`

`Extend(G, I): GrpHecke, RngOrdIdl -> GrpHecke`

`Extend(G, I, oo): GrpDrchNF, RngOrdIdl, SeqEnum -> GrpDrchNF`

`Extend(G, I, oo): GrpHecke, RngOrdIdl, SeqEnum -> GrpHecke`

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 $(I/J)^\star$. 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.

43.1.3 Predicates on Group Elements

`IsTrivial(chi): GrpDrchNFElt -> BoolElt`

`IsTrivial(psi): GrpHeckeElt -> BoolElt`

Returns whether the given character corresponds to the trivial element in the character group.

`IsTrivialOnUnits(chi): GrpDrchNFElt -> BoolElt`

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): GrpDrchNFElt -> BoolElt`

Returns whether a Dirichlet character χ has $\chi(-1) = -1$.

`IsEven(chi): GrpDrchNFElt -> BoolElt`

Returns whether a Dirichlet character χ has $\chi(-1) = +1$.

`IsTotallyEven(chi): GrpDrchNFElt -> BoolElt`

Returns whether a Dirichlet character χ has $\chi_p(-1) = +1$ for each χ_p in its decomposition.

`IsPrimitive(chi): GrpDrchNFElt -> BoolElt`

`IsPrimitive(psi): GrpHeckeElt -> BoolElt`

Returns whether a Dirichlet character is primitive, that is, whether its conductor and modulus are equal.

43.1.4 Passing between Dirichlet and Hecke Characters

`HeckeLift(chi): GrpDrchNFElt -> GrpHeckeElt, GrpHecke`

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): GrpHeckeElt -> GrpDrchNFElt`

Given a Hecke character on the ideals of the integer ring of a number field, this function returns the Dirichlet restriction of it on the field elements.

`NormInduction(K, chi): FldNum, GrpDrchElt -> GrpHeckeElt`

Given a Dirichlet character χ over the rationals (as `FldRat`) induce it to a Hecke character over the number field K. That is, find ψ with $\psi(a) = \chi(\mathrm{N}a)$ with ψ primitive.

`QuadraticCharacter(e): FldNumElt -> GrpHeckeElt`

Given a nonzero field element e over an absolute number field, return the associated quadratic Hecke character (which will be trivial when e is a square).

Example: Dirichlet Characters over the Rationals (ex-853a78)

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 ]
```

(continues on next page)

(continued from previous page)

```
> 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. In terms of class field theory, this can viewed as saying that the global Artin map relates -1 to complex conjugation.

Example: Basic Character Operations (ex-b4977b)

This example gives some basic character operations.

```
> K := QuadraticField(-23);
> p3 := Factorization(3*Integers(K))[1][1];
> G1 := HeckeCharacterGroup(p3^1);
> G2 := HeckeCharacterGroup(p3^2);
> G3 := HeckeCharacterGroup(p3^3);
> assert Extend(G1.1,G3) eq G3.1^9;
> assert Restrict(G2.1^3,G1) eq G1.1;
> assert Extend(G1.1,p3^3) eq Extend(G2.1^3,p3^3);
> assert Restrict(Restrict(G3.1^3,p3^2)^3,p3) eq G1.1;
> assert G1 eq HilbertCharacterSubgroup(G1);
> assert Restrict(HilbertCharacterSubgroup(G2),G1) eq G1;
> assert Restrict(sub<G2|[G2.1^3]>,p3) eq G1;
> assert Extend(G2,G3) eq sub<G3|[G3.1^3]>;
> assert Extend(G1,p3^2) eq Restrict(sub<G3|[G3.1^9]>,G2);
```

```
> chi := KroneckerCharacter(-7);
> K := QuadraticField(5);
> theta := NormInduction(K,chi);
> Type(theta);
GrpHeckeElt
> Modulus(theta); assert IsPrimitive(theta);
Principal Prime Ideal, Generator: 7
[ 1, 2 ] // places at infinity
> cbrt := Parent(theta).1;
> assert cbrt^3 eq theta; // Order(cbrt) is 6
> SetTargetRing(~cbrt,GF(13)!2); // choice of gen
> cbrt(K.1); // now an element of GF(13)
10
> DirichletRestriction(cbrt)(K.1);
-zeta_6 + 1
```

DirichletCharacter(I, B): RngOrdIdl, Tup -> GrpDrchNFElt, GrpDrchNF

DirichletCharacter(I, oo, B): RngOrdIdl, SeqEnum, Tup -> GrpDrchNFElt, GrpDrchNF

DirichletCharacter(G, B): GrpDrchNF, Tup -> GrpDrchNFElt, GrpDrchNF

DirichletCharacter(I, L): RngOrdIdl, List -> GrpDrchNFElt, GrpDrchNF

DirichletCharacter(I, oo, L): RngOrdIdl, SeqEnum, List -> GrpDrchNFElt, GrpDrchNF

DirichletCharacter(G, L): GrpDrchNF, List -> GrpDrchNFElt, GrpDrchNF

HeckeCharacter(I, B): RngOrdIdl, Tup -> GrpHeckeElt

HeckeCharacter(I, oo, B): RngOrdIdl, SeqEnum, Tup -> GrpHeckeElt

HeckeCharacter(G, B): GrpHecke, Tup -> GrpHeckeElt

HeckeCharacter(I, L): RngOrdIdl, List -> GrpHeckeElt

HeckeCharacter(I, oo, L): RngOrdIdl, SeqEnum, List -> GrpHeckeElt

HeckeCharacter(G, L): GrpHecke, List -> GrpHeckeElt

```
RequireGenerators: BoolElt                    Default: true
```

Given either an ideal (and also possibly a set of real infinite places) or a DirichletGroup, and a list/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 a torsion 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 can now be an ideal of the field, and RequireGenerators demands that these generate the RayClassGroup of the ideal.

Example: Dirichlet Character with Prescribed Values (ex-80bd7f)

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 parameter.

```
> _<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): GrpDrchNFElt -> GrpDrchNFElt

CentralCharacter(psi): GrpHeckeElt -> GrpDrchNFElt

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 $\mathbb{Q}$ 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: Central Character of a Dirichlet Character (ex-3ed3f9)

```
> 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): GrpDrchElt -> GrpDrchNFElt

`DirichletCharacterOverQ(chi): GrpDrchNFElt -> GrpDrchElt`

These are utility functions to pass between the two types of Dirichlet character over $\mathbb{Q}$ in Magma. The first takes a Dirichlet character over the `Rationals()` and returns one over the rationals as a number field (or more precisely over the first cyclotomic field, which unlike `QNF()` is unique in Magma), and the second reverses this.

Example: Dirichlet Characters over Q and Number Fields (ex-5e238b)

```
> G := DirichletGroup(16*3^2*5^2*7*11, CyclotomicField(2^6*3*5));
> #G;
57600
> repeat chi := Random(G); until Order(chi) eq 30;
> psi := DirichletCharacterOverNF(chi);
> Order(psi);
30
> #Parent(psi)`ambient;
57600
> &and[chi(p) eq psi(p) : p in PrimesUpTo(1000)];
true
> DirichletCharacterOverQ(psi) eq chi;
true
```

43.1.5 L-functions of Hecke Characters

Given a primitive Hecke character ψ, one can define the associated L-function as

$$L(\psi, s) = \prod_{\mathfrak{p}} \left(1 - \psi(\mathfrak{p})/\mathbb{N}\mathfrak{p}^s\right)^{-1},$$

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: L-Function of a Hecke Character (ex-d505ea)

```
> _<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); // approx with old CheckFunctionalEquation
4042
> CFENew(L);
-3.492459655E-10
```

43.1.6 Hecke Grössencharacters and their L-functions

Computations with Grössencharacters and their L-functions can now be done in Magma. 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, that is, 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 methods used are described in [Wat11].

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 is 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): GrpHeckeElt, GrpDrchNFElt, SeqEnum -> GrossenChar`

`RawEval(I, GR): RngOrdFracIdl, GrossenChar -> FldNumElt, FldCycElt, FldCycElt`

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 (note that [Wat11, §5.2] makes the reciprocal choice). 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.

For cyclotomic fields in Magma there is a specific order of the internal embeddings, while for general CM fields the user can obtain the ordering via the output of `Conjugates` or evaluating at `InfinitePlaces`.

`Grossencharacter(psi, T): GrpHeckeElt, SeqEnum -> GrossenChar`

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(GR): GrossenChar -> RngOrdIdl, SeqEnum`

`Modulus(GR): GrossenChar -> RngOrdIdl, SeqEnum`

`IsPrimitive(GR): GrossenChar -> BoolElt`

`AssociatedPrimitiveGrossencharacter(psi): GrossenChar -> GrossenChar`

The conductor of the Grössencharacter is the conductor of the quotient of the `DirichletRestriction` of the Hecke part divided by its Dirichlet 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.

`Components(GR): GrossenChar -> Assoc`

Given a Grössencharacter, return the components of its Hecke part divided by the reciprocal of its Dirichlet part.

`Component(GR, P): GrossenChar, RngOrdIdl -> GrpDrchNFEl`

`Component(GR, oo): GrossenChar, RngIntElt -> GrpDrchNFEl`

`Component(GR, P): GrossenChar, PlcNumElt -> GrpDrchNFEl`

Given a Grössencharacter and a prime ideal or place (possibly specified by an integer for an infinite place), return the corresponding Dirichlet character component. For a place that it is not ramified, the trivial character of the integer ring is returned.

`Extend(GR, I): GrossenChar, RngOrdIdl -> GrossenChar`

`Restrict(GR, I): GrossenChar, RngOrdIdl -> GrossenChar`

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(GR): GrossenChar -> GrpDrchNFElt`

Compute the central character (down to $\mathbb{Q}$) of a Grössencharacter, normalizing the result to be weight 0, and returning it as a Dirichlet character (over $\mathbb{Q}$ as a number field).

`GrossenTwist(GR, D): GrossenChar, List -> GrossenChar, GrpHecke`

```
Hilbert: BoolElt                    Default: false
```

Given a Grossencharacter Y and a list D of tuples $\langle a, r\rangle$, find a twist Ψ of Y (by a Hecke character of the same modulus as Y) such that $\Psi(a) = r$ for all data tuples. This operates by dividing the given r-values by the evaluations of Y at the given ideals, and using `HeckeCharacter` on the resulting cyclotomic data. The intrinsic also returns a group of Hecke characters that corresponds to a kernel. The given values r should be coercible into the complex numbers, and a numerical matching is made. The a should be ideals or field elements. The `Hilbert` parameter restricts to twisting by characters of trivial modulus.

`TateTwist(GR, n): GrossenChar, RngIntElt -> GrossenChar`

`TateTwist(psi, n): GrpHeckeElt, RngIntElt -> GrossenChar`

Given a Grossencharacter or a Hecke character, return its Tate twist as a Grossencharacter. Note that the field need not be CM in the latter case.

Example: Grossencharacters on the Gaussian Field (ex-42864c)

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); CFENew(L);
1.5777218104420236108234571305 7E-30
> CentralValue(L);
0.65551438857302995261620989747 5
> CentralValue(LSeries(EllipticCurve("32a")));
0.65551438857302995261620989747 3
```

Example: Canonical Grossencharacter of Q(sqrt(-23)) (ex-b4291f)

An example with the canonical Grössencharacter for $K = \mathbb{Q}(\sqrt{-23})$. 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
```

(continues on next page)

(continued from previous page)

```
> CFENew(LSeries(GR));
4.733165431326070832470371391 70E-30
> 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; in general, this `extension_field` attribute is a capitulation field for K (thus K itself when the class number is 1), and the values of the Grössencharacter can be further twisted by cyclotomic factors.

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
-0.14015763895624666594418088012 0 - 1.40725116316960648195556086783*i
> GR2 := Grossencharacter(HG.1, DG.1^11, [[1,0]]); // psi over zeta_11
> GR2(i2);
-0.14015763895624666594418088012 0 - 1.40725116316960648195556086783*i
> RawEval(i2,GR2);  // first value is in the cubic extension of K
H.1
1
zeta_33^4
> CFENew(LSeries(GR2));
-7.09974814698910624870555708755E-30
```

Example: Grossencharacter with Vanishing Central Value `(ex-7d182f)`

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 $\mathbb{Q}(\sqrt{-59})$, 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);
> CFENew(L);
0.0000000000000000000000000000000
> Sign(L);
1.0000000000000000000000000000000
> CentralValue(L);
3.5185802675986407547501792565 0E-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));
3.5185802675986407547501792565 0E-30
```

Example: Grossencharacter Central Value and Gamma Periods `(ex-38ad25)`

An example with $\mathbb{Q}(\zeta_5)$, 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, ... >;
> CFENew(L); // functional equation works
0.00000000000000000000000000000
> 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);
> PI := Pi(RealField());
> CentralValue(L); // now recognise as a product via logs and LLL
0.74985924643337212300558568330 0
> A := [ Gamma(1/5), Gamma(2/5), Gamma(3/5), Gamma(4/5), 5, PI, $1 ];
> LOGS := [ ComplexField() ! Log(x) : x in A ];
> IntegerRelation(LOGS);
[ -14, 2, -2, 14, 15, -4, 4 ]
```

Example: Twisting a Grossencharacter by a Hilbert Character (ex-a39eeb)

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 _:=CFENew(L) : L in Ls ]; // force Sign computation
[ 0.99999999999999999999999999997, 1.00000000000000000000000000000,
  0.99999999999999999999999999997, 1.00000000000000000000000000000 ]
```

Example: Grossencharacters of Trivial Conductor (ex-03fd6d)

A final example with characters of trivial conductor, here of type $[2,0]$ in $\mathbb{Q}(\sqrt{-23})$.

```
> 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
1.2381910021242604040079438 4795
> Evaluate(LSeries(GR*HG.1), 2); // twist by nontrivial Hecke char
0.67033720866583940374792 2477469
> Evaluate(LSeries(GR*HG.1^2), 2);
1.06110583266449728309907405960
```

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);
```

(continues on next page)

(continued from previous page)

```
> 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];
> IntegerRelation(LOGS);
[ 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] ];
24723927.962642907904478305429424516264331853471961573093155911 28
> 3^2 * 23^7 / Pi(RealField())^6 * (e1*e2*e3)^2;
24723927.962642907904478305429424516264331853471961573093155911 28
```

Example: Grossencharacter on the Eighth Cyclotomic Field (ex-105c44)

An example from [vGvS93, §8.8]. Here the Grössencharacter is on an ideal of norm 2^4 in the cyclotomic field $\mathbb{Q}(\zeta_8)$. 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);
> CFENew(L);
6.3108872417680944432938285 2226E-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
```

Example: Grossencharacter from a Hypergeometric Motive (ex-d12cc1)

This examples exhibits the use of `GrossenTwist`, and links a Grössencharacter to a hypergeometric datum. The chosen t-value is such that the degree 3 L-function is imprimitive, and it splits as $L(\chi_{12}, s+1)L(\Psi, s)$, where Ψ is defined over $\mathbb{Q}(\sqrt{-84})$, and has trivial modulus and $[2, 0]$ ∞-type; there are still four such characters (the class number is 4), and we want the one with $\Psi(\mathfrak{p}_2) = 2$ and $\Psi(\mathfrak{p}_3) = -3$. We then check the degree 3 Euler factors on all good primes up to 100.

```
> P := PrimesInInterval(5,100);
> H := HypergeometricData([2,3],[1,6]);
> t := -27;
> ZT := Translate(LSeries(KroneckerCharacter(12)),1);
> K := QuadraticField(-84);
> DATA2 := <Factorization(2*Integers(K))[1][1],2>;
> DATA3 := <Factorization(3*Integers(K))[1][1],-3>;
> G := HeckeCharacterGroup(1*Integers(K));
> GR := Grossencharacter(G.0,[[2,0]]);
> LGR := LSeries(GrossenTwist(GR,[* DATA2, DATA3 *]));
```

(continues on next page)

(continued from previous page)

```
> PROD := LGR*ZT;
> assert &and[EulerFactor(PROD,p : Integral) eq
>     EulerFactor(H,t,p) : p in P];
```

And another (simpler) example of GrossenTwist.

```
> K := QuadraticField(-1);
> H := HeckeCharacterGroup(7*Integers(K));
> psi := H.1; // order 12
> GR := Grossencharacter(psi,[[1,0]]);
> TW := GR*psi^7;
> P := PrimesUpTo(100,K);
> D :=[* <p,TW(p)> : p in P | Gcd(14,Norm(p)) eq 1 *];
> assert GrossenTwist(GR,D) eq TW;
```

43.1.7 Local Root Numbers

One can also compute the local root numbers of Hecke characters and Grössencharacters at prime ideals over the field of definition and at places at infinity. Multiplying these together gives the global root number.

RootNumber(GR, P): GrossenChar, RngOrgIdl -> FldComElt

RootNumber(psi, P): GrpHeckeElt, RngOrgIdl -> FldComElt

RootNumber(GR, P): GrossenChar, PlcNumElt -> FldComElt

RootNumber(psi, P): GrpHeckeElt, PlcNumElt -> FldComElt

```
Precision: RngIntElt                    Default:
```

Given a Hecke character or Grössencharacter and a place or a prime ideal, return the local root number as a complex number.

RootNumber(GR, p): GrossenChar, RngIntElt -> FldComElt

RootNumber(psi, p): GrpHeckeElt, RngIntElt -> FldComElt

```
Precision: RngIntElt                    Default:
```

Given a Hecke character or Grössencharcter and a rational prime p, compute the (induced) root number at p.

RootNumbers(GR): GrossenChar -> SeqEnum

RootNumbers(psi): GrossenChar -> SeqEnum

```
Precision: RngIntElt                    Default:
AA       : BoolElt                      Default: false
```

Given a Hecke character or Grössencharacter, return the local root numbers at bad places (including infinite ones) as an array of tuples with places and complex approximations given. If the AA vararg is set, the answer is returned as an associative array.

RootNumber(GR): GrossenChar -> SeqEnum

RootNumber(psi): GrossenChar -> SeqEnum

```
Precision: RngIntElt                    Default:
```

Given a Hecke character or Grössencharacter, return the global root number.

Example: Local Root Numbers for Curve 27a (ex-a7b80a)

We compute the local root numbers of the Grössencharacter over $\mathbb{Q}(\sqrt{-3})$ for the elliptic curve 27a, and some of its powers.

```
> G := Grossencharacter(EllipticCurve("27a")); G;
Grossencharacter G of type [[ 1, 0 ]] for Hecke-Dirichlet pair
(1,$.1*$.2^2) with modulus of norm 9 over Quadratic Field x^2 + 3
> K := NumberField(Order(Modulus(G)));
> I := Ideal(Decomposition(K,3)[1][1]);
> RootNumber(G,I);
6.80316131261355894597927551960E-31 + 1.00000000000000000000000000000*i
> RootNumber(G,InfinitePlaces(K)[1]);
-1.00000000000000000000000000000*i
> RootNumber(G); // product of the two previous
1.00000000000000000000000000000 - 6.80316131261355894597927551960E-31*i
> // over Q, the root numbers get induced to -1 and -1
> RootNumber(G,3); // at the rational prime 3
-1.00000000000000000000000000000 + 6.80316131261355894597927551960E-31*i
> RootNumber(HodgeStructure(G)); // at oo
-1
> for k in [1..12] do k,Real(RootNumber(G^k)); end for;
1 1.00000000000000000000000000000
2 1.00000000000000000000000000000
3 1.00000000000000000000000000000
4 1.00000000000000000000000000000
5 -1.00000000000000000000000000000
6 1.00000000000000000000000000000
7 1.00000000000000000000000000000
8 1.00000000000000000000000000000
9 -1.00000000000000000000000000000
10 1.00000000000000000000000000000
11 -1.00000000000000000000000000000
12 1.00000000000000000000000000000
```

Example: Rohrlich's Root Number Exercise (ex-f91aec)

This example comes from Exercise 5.5 in Rohrlich's PCMI lectures [Roh].

We take the Grössencharacter corresponding to the elliptic curve 49a, and then twist it over $\mathbb{Q}(\sqrt{-7})$ by $-118 - 18\sqrt{-7}$. This disturbs the root number and the resulting L-function is no longer self dual, but it still vanishes at the central point.

```
> GR := Grossencharacter(EllipticCurve("49a"));
> K := NumberField(Order(Modulus(GR))); K; // ensure same field
Quadratic Field with defining polynomial x^2 + 7 over the Rational Field
> psi := QuadraticCharacter(-118-18*K.1);
> RootNumber(GR);
1.00000000000000000000000000000 + 1.04356488711804586030321101838E-30*i
> RootNumber(psi);
1.00000000000000000000000000000 - 3.55311676760811128464401934867E-32*i
> RootNumber(GR*psi);
0.94304192019289721464894137352 - 0.33267391956523024173849624956*i
> L := LSeries(GR*psi);
> CentralValue(L); // zero
2.91343312836088675607917006164E-30
> Sign(L); // root of 253*x^4 - 394*x^2 + 253
0.94304192019289721464894137361 - 0.33267391956523024173849624956*i
```

43.1.8 Grössencharacters and Elliptic Curves

There are also functions that transform between elliptic curves over $\mathbb{Q}$ that have complex multiplication (over an imaginary quadratic field), and Grössencharacters associated to them.

`Grossencharacter(E): CrvEll -> GrossenChar`

Given an elliptic curve over $\mathbb{Q}$ with complex multiplication by an imaginary quadratic order, return the associated Grössencharacter.

`EllipticCurve(GR): GrossenChar -> CrvEll`

Given a suitable Grössencharacter, in particular of ∞-type $[1, 0]$ over an imaginary quadratic field, with the underlying character of order 2, return an elliptic curve in the associated isogeny class.

Example: Grossencharacters of CM Elliptic Curves (ex-e6dfed)

```
> E := EllipticCurve("49a"); // cm by -7
> G := Grossencharacter(QuadraticTwist(E,5*29));
> Conductor(QuadraticTwist(EllipticCurve(G),5*29));
49
> //
> E := EllipticCurve([2^2 * 3 * 5^3 * 7 * 11^2 * 13^3, 0]);
> G := Grossencharacter(E);
> assert IsIsogenous(E,EllipticCurve(G));
> //
> E := EllipticCurve([0, 2 * 3^5 * 5^3 * 7^4 * 11^2 * 13^5]);
> G := Grossencharacter(E);
> assert IsIsogenous(E,EllipticCurve(G));
> //
> K := QuadraticField(-163);
> p := Factorization(163*Integers(K))[1][1];
> psi := HeckeCharacterGroup(p).0;
> G := Grossencharacter(psi,[[1,0]]);
> E := EllipticCurve(G);
> b, cm := HasComplexMultiplication(E);
> assert b; assert cm eq -163;
> Conductor(E);
26569
```

43.2 Bibliography

- [Roh] D. E. Rohrlich. "Root Numbers." In Arithmetic L-functions. 18, 353–448.
- [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.
- [Wat11] M. Watkins. Computing with Hecke Grössencharacters. *Publications mathématiques de Besançon*, 2011/1:119–135, 2011.

Chapter 44

Algebraically Closed Fields

44.1 Introduction

Magma contains a system [Ste, 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.

44.2 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 $\mathbb{Q}$, or a rational function field over a finite field or $\mathbb{Q}$.

44.3 Creation of Structures

`AlgebraicClosure(K): Fld -> FldAC`

Create the algebraic closure **A** of the field K. Currently K may be $\mathbb{Q}$, a finite field, or a rational function field over a finite field or the rational field.

`AlgebraicClosure() -> FldAC`

Create the algebraic closure **A** of the rational field $\mathbb{Q}$.

`AssignNamePrefix(A, S): FldAC, MonStgElt`

(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).

44.4 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).

44.4.1 Coercion

`A ! a: FldAC, RngElt -> FldACElt`

Given an algebraically closed 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): FldAC -> FldACElt`

`Identity(A): FldAC -> FldACElt`

`Zero(A): FldAC -> FldACElt`

`Representative(A): FldAC -> FldACElt`

These generic functions create `A!1`, `A!1`, `A!0`, and `A!0`, respectively.

44.4.2 Roots

`Roots(f): RngUPolElt -> [ < FldACElt, RngIntElt> ]`

`Roots(f, A): RngUPolElt, FldAC -> [ < FldACElt, RngIntElt> ]`

`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): RngIntElt, FldAC -> FldACElt`

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): FldACElt -> FldACElt`

`Sqrt(a): FldACElt -> FldACElt`

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): FldACElt -> BoolElt`

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): FldACElt, RngIntElt -> FldACElt`

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): FldACElt, RngIntElt -> BoolElt, FldACElt`

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.

44.4.3 Variables

`A . i: FldAC, RngIntElt -> FldACElt`

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: Creating Roots in the Algebraic Closure (ex-f1defe)

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;
```

(continues on next page)

(continued from previous page)

```
2
> cube3^3;
3
```

Example: Swinnerton-Dyer Polynomials (ex-681d9c)

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 $\mathbb{Z}[x]$ and are irreducible over $\mathbb{Z}$. 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 $\mathbb{Z}[x]$.

```
> 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 $\mathbb{Z}$, but they are no longer difficult to factor using van Hoeij's new algorithm (see Example Factoring Swinnerton-Dyer Polynomials (ex-d3c652)).

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 $\mathrm{GSD}_Q \in \mathbb{Z}[x]$, is irreducible over $\mathbb{Z}$, 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])]);
17710807204306291616851589788921525994 56 11
```

Example: Puiseux Expansions over an Algebraic Closure (ex-efef60)

This example shows how one can compute Puiseux expansions over an algebraic closure. The `PuiseuxExpansion(f, n): RngUPolElt, RngIntElt → SeqEnum[RngSerPuisElt]` 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),
```

(continues on next page)

(continued from previous page)

```
    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)
]
```

44.5 Related Structures

Category(A): FldAC -> Cat

Parent(A): FldAC -> PowerStructure

Centre(A): FldAC -> AC

PrimeRing(A): FldAC -> AC

PrimeField(A): FldAC -> AC

`FieldOfFractions(A): FldAC -> AC`

44.6 Properties

`BaseField(A): FldAC -> Fld`

Return the base field over which A is defined.

`Rank(A): FldAC -> RngIntElt`

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): FldAC, RngIntElt -> RngIntElt`

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): FldAC -> RngIntElt`

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): FldAC -> RngMPolRes`
`QuotientRing(A): FldAC -> RngMPolRes`

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): FldAC -> RngMPol`

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*.

44.7 Ring Predicates and Properties

`IsCommutative(A): FldAC -> BoolElt`

`IsUnitary(A): FldAC -> BoolElt`

`IsFinite(A): FldAC -> BoolElt`

`IsOrdered(A): FldAC -> BoolElt`

`IsField(A): FldAC -> BoolElt`

`IsEuclideanDomain(A): FldAC -> BoolElt`

`IsPID(A): FldAC -> BoolElt`

`IsUFD(A): FldAC -> BoolElt`

```
IsDivisionRing(A): FldAC -> BoolElt

IsEuclideanRing(A): FldAC -> BoolElt

IsPrincipalIdealRing(A): FldAC -> BoolElt

IsDomain(A): FldAC -> BoolElt

A eq B: FldAC, Rng -> BoolElt

A ne B: FldAC, Rng -> BoolElt

Characteristic(A): FldAC -> FldACElt
```

44.8 Element Operations

44.8.1 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: FldACElt -> FldACElt

- a: FldACElt -> FldACElt

a + b: FldACElt, FldACElt -> FldACElt

a - b: FldACElt, FldACElt -> FldACElt

a * b: FldACElt, FldACElt -> FldACElt

a / b: FldACElt, FldACElt -> FldACElt

a ^ k: FldACElt, RngIntElt -> FldACElt

a +:= b: FldACElt, FldACElt -> FldACElt

a -:= b: FldACElt, FldACElt -> FldACElt

a *:= b: FldACElt, FldACElt -> FldACElt
```

44.8.2 Equality and Membership

44.8.3 Parent and Category

```
Parent(a): FldACElt -> AC

Category(a): FldACElt -> Cat
```

44.8.4 Predicates on Ring Elements

```
IsZero(a): FldACElt -> BoolElt
```

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): FldACElt -> BoolElt`

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): FldACElt -> BoolElt`

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: FldACElt, FldACElt -> BoolElt`

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: FldACElt, FldACElt -> BoolElt`

`a in A: FldACElt, Rng -> BoolElt`

`a notin A: FldACElt, Rng -> BoolElt`

`IsNilpotent(a): FldACElt -> BoolElt`

`IsIdempotent(a): FldACElt -> BoolElt`

`IsUnit(a): FldACElt -> BoolElt`

`IsZeroDivisor(a): FldACElt -> BoolElt`

`IsRegular(a): FldAC -> BoolElt`

`IsIrreducible(a): FldACElt -> BoolElt`

`IsPrime(a): FldACElt -> BoolElt`

44.8.5 Minimal Polynomial, Norm, and Trace

`MinimalPolynomial(a): FldACElt -> RngUPolElt`

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): FldACElt -> FldACElt`

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): FldACElt -> FldACElt`

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-1} in M, where M is the irreducible minimal polynomial of a returned by *MinimalPolynomial*(a) and $n = \text{Degree}(M)$. Consequently, a simplification of the field may occur, but the return value is invariable.

`Conjugates(a): FldACElt -> [ FldACElt ]`

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: Conjugate Elements of an Algebraic Closure (ex-e31b1f)

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; // depends on choice of square roots
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
```

44.9 Simplification

The following procedures allow one to simplify an algebraically closed field so that it is a true field.

`Simplify(A): FldAC`

```
Partial: BoolElt                    Default: false
```

(Procedure.) Simplify the internal representation of A in place, so that the ideal of its defining polynomials is explicitly maximal. Elements of A computed earlier stay valid, though they may print differently.

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): FldAC`

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.

44.10 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): FldAC -> RngUPolRes`

`AbsoluteQuotientRing(A): FldAC -> RngUPolRes`

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): FldAC`

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.

`NumberField(A): FldAC -> FldNum, Map`

```
Absolute: BoolElt                   Default: false
```

The number field N isomorphic to the subfield of A generated by the elements introduced so far, and a bidirectional map $f : A \to N$. By default N is a tower with one extension for each variable of A. If the parameter `Absolute` is set to `true`, then N is an absolute number field instead. A is simplified and pruned in place (see *Simplify*`(A)` and *Prune*`(A)` above). The map f is invalidated when A changes later, for example when a new root is computed or A is pruned again.

`NumberField(a): FldACElt -> FldNum, Map`

The number field N generated by a over the base field of its parent A, and the embedding $N \to A$ sending $N.1$ to a. N is defined by the minimal polynomial of a (see *MinimalPolynomial*), so A is not simplified as a whole.

`Absolutize(A): FldAC`

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: Field of Definition of the Cyclic-6 Variety `(ex-32dfbd)`

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 $\mathbb{Q}$ and compute its variety over A, the algebraic closure of $\mathbb{Q}$.

```
> 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);
30
> 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
]
```

(continues on next page)

(continued from previous page)

```
> 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: Splitting Field of a Degree 8 Polynomial (ex-862d19)

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);
```

(continues on next page)

(continued from previous page)

```
> #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
```

Example: Splitting Field of a Quartic as a Number Field (ex-d2e9ca)

In this example we present the splitting field of $x^4 - x - 1$, whose Galois group is S_4, as a number field.

```
> P<x> := PolynomialRing(RationalField());
> f := x^4 - x - 1;
> #GaloisGroup(f);
```

(continues on next page)

(continued from previous page)

```
24
> A := AlgebraicClosure();
> r := Roots(f, A);
> #r;
4
```

By default `NumberField` presents the splitting field as a tower, with one small defining polynomial per level: a root of f, a root of the residual cubic above it, then a root of the residual quadratic. The map m sends the roots of f in A to roots of f in N.

```
> N, m := NumberField(A);
> AbsoluteDegree(N);
24
> DefiningPolynomial(N);
$.1^2 + (r2 + r1)*$.1 + r2^2 + r1*r2 + r1^2
> DefiningPolynomial(BaseField(N));
$.1^3 + r1*$.1^2 + r1^2*$.1 + r1^3 - 1
> DefiningPolynomial(BaseField(BaseField(N)));
x^4 - x - 1
> Evaluate(f, m(r[1][1]));
0
```

With `Absolute` the same field is defined by a single polynomial of degree 24 in a primitive element. The size of that polynomial grows quickly with the degree of the field, which is why the tower is the default.

```
> Na, ma := NumberField(A : Absolute);
> DefiningPolynomial(Na);
x^24 + 80*x^20 + 340*x^18 + 7520*x^16 - 23120*x^14 + 1241022*x^12 -
    462400*x^10 + 48052400*x^8 - 61067260*x^6 + 756753632*x^4 +
    2074608720*x^2 + 1215289321
> Evaluate(f, ma(r[1][1]));
0
```

Example: Splitting Field of Degree 720 as a Tower `(ex-fe6f00)`

For larger splitting fields the tower is the only practical presentation. The splitting field of $x^6 - x - 1$, whose Galois group is S_6, has degree 720. As a tower with one extension for each variable of A it takes the time shown below. The `Absolute` presentation is orders of magnitude more expensive, and its defining polynomial is far too large to print.

```
> P<x> := PolynomialRing(IntegerRing());
> f := x^6 - x - 1;
> #GaloisGroup(f);
720
> A := AlgebraicClosure();
> r := Roots(f, A);
> time N, m := NumberField(A);
Time: 20.140
> Degree(N), AbsoluteDegree(N);
2 720
> Evaluate(f, m(r[1][1]));
0
```

44.11 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, 289–290. Linz, 1985. Springer.
- [Ste] A. Steel. "A new scheme for computing with algebraically closed fields." 491–505.
- [Ste10] Allan K. Steel. Computing with algebraically closed fields. *J. Symb. Comput.*, 45(3):342–372, March 2010. URL: http://dx.doi.org/10.1016/j.jsc.2009.09.005, doi:10.1016/j.jsc.2009.09.005.

Chapter 45

Rational Function Fields

45.1 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 $\mathbb{Z}$). Note that R itself need not be a field. Thus it is possible, for example, to create the rational function field $K = \mathbb{Z}(t)$ which is mathematically equal to $\mathbb{Q}(t)$ of course, but will be represented slightly differently. A fraction in $\mathbb{Q}(t)$ will have a monic denominator (and the coefficients of both the numerator and denominator may be non-integral), while a fraction in $\mathbb{Z}(t)$ will have a positive denominator (and the coefficients of both the numerator and denominator will be integral). Thus the fractions $(3t+2)/(4t-2) \in \mathbb{Z}(t)$ and $((3/4)t+1/2)/(t-1/2) \in \mathbb{Q}(t)$ 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 $\mathbb{Z}(t)$ instead of $\mathbb{Q}(t)$) since arithmetic is much faster, but the use of a field of fractions for the coefficient ring may be more desirable for output purposes.

45.2 Creation Functions

45.2.1 Creation of Structures

`FunctionField(R): Rng -> FldFunRat`

`RationalFunctionField(R): Rng -> FldFunRat`

```
Global: BoolElt                    Default: true
```

Create the field $\mathbb{F}$ 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): Rng, RngIntElt -> FldFunRat`

`RationalFunctionField(R, r): Rng, RngIntElt -> FldFunRat`

```
Global: BoolElt                    Default: false
```

Create the field $\mathbb{F}$ 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): RngUPol -> FldFunRat`

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.

45.2.2 Names

`AssignNames(~F, s): FldFunRat, [ MonStgElt ]`

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 $1 \leq i \leq \#s$); 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): FldFunRat, RngIntElt -> FldFunRatElt`

Given a function field F, return the i-th indeterminate of F (as an element of F).

45.2.3 Creation of Elements

`F ! [a, b]: FldFunRat, RngUPolElt, RngUPolElt -> FldFunRatElt`

`elt< F | a, b >: FldFunRat, RngUPolElt, RngUPolElt -> FldFunRatElt`

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: FldFunRat, FldElt -> FldFunRatElt`

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: FldFunRat, RngIntElt -> FldFunRatElt`

The i-th generator for the field of fractions K of R over the coefficient ring of R.

`One(F): FldFunRat -> FldFunRatElt`

`Identity(F): FldFunRat -> FldFunRatElt`

`Zero(F): FldFunRat -> FldFunRatElt`

`Representative(F): FldFunRat -> FldFunRatElt`

Example: Creating Rational Function Field Elements (ex-64b44d)

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)
```

45.3 Structure Operations

45.3.1 Related Structures

IntegerRing(F): FldFunRat -> RngPol

RingOfIntegers(F): FldFunRat -> RngPol

Given the rational function field F this returns the polynomial ring from which F was constructed as its field of fractions.

BaseRing(F): FldFunRat -> Rng

CoefficientRing(F): FldFunRat -> Rng

The coefficient ring of the (ring of integers of) the rational function field F.

Rank(F): FldFunRat -> RngIntElt

The rank (number of indeterminates) of the rational function field F.

ValuationRing(F): FldFunRat -> RngVal

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): FldFunRat, RngUPolElt -> RngVal

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): FldFunRat -> Cat

Parent(R): FldFunRat -> Pow

PrimeRing(R): FldFunRat -> Rng

45.3.2 Invariants

Characteristic(F): FldFunRat -> FldFunRatElt

45.3.3 Ring Predicates and Booleans

IsCommutative(F): FldFunRat -> BoolElt

IsUnitary(F): FldFunRat -> BoolElt

IsFinite(F): FldFunRat -> BoolElt

IsOrdered(F): FldFunRat -> BoolElt

IsField(F): FldFunRat -> BoolElt

IsEuclideanDomain(F): FldFunRat -> BoolElt

```
IsPID(F): FldFunRat -> BoolElt

IsUFD(F): FldFunRat -> BoolElt

IsDivisionRing(F): FldFunRat -> BoolElt

IsEuclideanRing(F): FldFunRat -> BoolElt

IsPrincipalIdealRing(F): FldFunRat -> BoolElt

IsDomain(F): FldFunRat -> BoolElt

F eq G: FldFunRat, Rng -> BoolElt

F ne G: FldFunRat, Rng -> BoolElt
```

45.3.4 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 >: FldFunRat, Rng -> Map`

`hom< P -> S | y1, ..., yn >: FldFunRat, Rng -> Map`

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: Homomorphism to a Number Field (ex-bca4bf)

In this example we map $\mathbb{Q}(x, y)$ into the number field $\mathbb{Q}(\sqrt[3]{2}, \sqrt{5})$ 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
```

45.4 Element Operations

45.4.1 Arithmetic

```
+ a: FldFunRatElt -> FldFunRatElt

- a: FldFunRatElt -> FldFunRatElt

a + b: FldFunRatElt, FldFunRatElt -> FldFunRatElt
```

`a - b: FldFunRatElt, FldFunRatElt -> FldFunRatElt`

`a * b: FldFunRatElt, FldFunRatElt -> FldFunRatElt`

`a / b: FldFunRatElt, FldFunRatElt -> FldFunRatElt`

`a ^ k: FldFunRatElt, RngIntElt -> FldFunRatElt`

45.4.2 Equality and Membership

`a eq b: FldFunRatElt, FldFunRatElt -> BoolElt`

`a ne b: FldFunRatElt, FldFunRatElt -> BoolElt`

`a in F: FldFunRatElt, FldFunRat -> BoolElt`

`a notin F: FldFunRatElt, FldFunRat -> BoolElt`

45.4.3 Numerator, Denominator, and Degree

`Numerator(f): FldFunRatElt -> RngElt`

`Numerator(f): RngUPolElt -> RngUPolElt`

`Numerator(f): RngMPolElt -> RngMPolElt`

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): FldFunRatElt -> RngElt`

`Denominator(f): RngUPolElt -> RngUPolElt`

`Denominator(f): RngMPolElt -> RngMPolElt`

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): FldFunRatElt -> RngIntElt`

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): FldFunRatElt -> RngIntElt`

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): FldFunRatElt -> RngIntElt`

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).

`Numerator(f, R): FldFunRatElt -> RngElt`

`Denominator(f, R): FldFunRatElt -> RngElt`

Return the numerator or denominator of f with respect to the ring of integers R of the rational function field F containing f. The ring R may be a polynomial ring or a valuation ring.

45.4.4 Predicates on Ring Elements

`IsZero(a): FldFunRatElt -> BoolElt`

`IsOne(a): FldFunRatElt -> BoolElt`

`IsMinusOne(a): FldFunRatElt -> BoolElt`

`IsNilpotent(a): FldFunRatElt -> BoolElt`

`IsIdempotent(a): FldFunRatElt -> BoolElt`

`IsUnit(a): FldFunRatElt -> BoolElt`

`IsZeroDivisor(a): FldFunRatElt -> BoolElt`

`IsRegular(a): FldFunRatElt -> BoolElt`

45.4.5 Evaluation

`Evaluate(f, r): FldFunRatUElt, RngElt -> FldFunRatUElt`

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): FldFunRatMElt, RngIntElt, RngElt -> FldFunRatMElt`

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.

`Evaluate(f, S): FldFunRatMElt, [RngElt] -> RngElt`

Given a multivariate rational function f in F return the result of evaluating the v-th variable at the v-th element of the sequence S which should be a sequence of elements coercible into the coefficient ring of the integers of F.

45.4.6 Decomposition

The following function provides an inverse operation to composing rational functions.

`Decomposition(f): FldFunRatUElt -> [[FldFunRatUElt]]`

All: BoolElt	Default: true

Given a rational function f defined over a field k, return all complete decompositions $[f_1, \ldots, f_r]$ of f such that $f(t) = f_r(\ldots(f_1(t))$. These decompositions are found, as explained in [ACvHS17], by computing the subfield lattice of $k(t)/k(f)$, which in turn, is found by first computing the principal subfields of this extension and then computing all intersections of principal subfields. These intersections are computed by joining partitions.

The routines in this package are those developed by Peter Fleischmann and Jonas Szutkoski.

Example: Decomposition of a Rational Function `(ex-221d21)`

We show a decomposition of a rational function and the corresponding evaluations which retrieve the function which was decomposed.

```
> k:=GF(3);
> F<t>:=FunctionField(k);
> f:=(t^10 + 2*t^8 + 2*t^4 + t^2)/(t^12 + 2*t^6 + 1);
> Decomposition(f);
[
    [
        t^2,
        t/(t^2 + 1),
        (2*t^3 + t^2 + 2*t)/(t^3 + 2)
    ],
    [
        t/(t^2 + 2),
        t^2,
        t/(t^3 + 1)
```

(continues on next page)

(continued from previous page)

```
    ],
    [
        t/(t^2 + 1),
        t^3 + 2*t,
        t^2
    ],
    [
        t/(t^2 + 1),
        t^2 + t,
        t^3 + t^2
    ],
    [
        t/(t^2 + 1),
        t^2 + 2*t,
        t^3 + t^2
    ],
    [
        t/(t^2 + 1),
        t^2,
        t^3 + t^2 + t
    ]
]
> Decomposition(f : All := false);
[
    [
        t^2,
        t/(t^2 + 1),
        (2*t^3 + t^2 + 2*t)/(t^3 + 2)
    ]
]
> Evaluate($1[1][3], Evaluate($1[1][2], $1[1][1])) eq f;
true
```

45.4.7 Derivative

`Derivative(f): FldFunRatUElt -> FldFunRatUElt`

Given a univariate rational function f, return the first derivative of f with respect to its variable.

`Derivative(f, k): FldFunRatUElt, RngIntElt -> FldFunRatUElt`

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): FldFunRatMElt, RngIntElt -> FldFunRatMElt`

Given a multivariate rational function f, return the first derivative of f with respect to variable number v.

`Derivative(f, v, k): FldFunRatMElt, RngIntElt, RngIntElt -> FldFunRatMElt`

Given a multivariate rational function f, return the k-th derivative of f with respect to variable number v. k must be non-negative.

45.4.8 Partial Fraction Decomposition

`PartialFractionDecomposition(f): FldFunRatUElt -> [ <RngUPolElt, RngIntElt, RngUPolElt> ]`

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): FldFunRatUElt -> [<RngUPolElt, RngIntElt, RngUPolElt>]

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: Partial Fraction Decomposition (ex-2ba727)

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;
[
```

(continues on next page)

(continued from previous page)

```
    <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
```

45.5 Padé-Hermite Approximants

45.5.1 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 [BL94a]. They are implemented in Magma as sequences rather than vectors.

45.5.2 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): SeqEnum -> RngIntElt`

`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: Maximum Degree of a Sequence (ex-ea74e8)

```
> 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): SeqEnum -> RngIntElt, RngIntElt`

```
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: Type of a Sequence (ex-8d75ca)

```
> 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): SeqEnum, RngIntElt -> SeqEnum`

```
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: Minimal Vector Sequence of Power Series (ex-67b928)

```
> 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: Minimal Vector Sequence of Polynomials (ex-d1c2b6)

```
> 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: Minimal Vector Sequence of Higher Order (ex-328015)

```
> 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: Minimal Vector Sequence with Distortion (ex-1d53c0)

```
> 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 0((S.1)^(p*v))
```

(continues on next page)

(continued from previous page)

```
>   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
]
```

45.5.3 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): SeqEnum, SeqEnum -> ModTupRngElt, SeqEnum, RngIntElt

```
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: Pade-Hermite Approximants (ex-05a6f0)

This example can be found on page 813 in [BL94a].

```
> 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
```

(continues on next page)

(continued from previous page)

```
>     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: Pade-Hermite Approximant with Distortion (ex-e98d49)

This example covers the example on page 815 in [BL94a].

```
> 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),
```

(continues on next page)

(continued from previous page)

```
    (      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: Degrees and Valuations of Approximants (ex-ea09f6)

This example considers the example on page 816 in [BL94a].

```
> 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): SeqEnum, RngIntElt -> ModTupRngElt, SeqEnum

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: Pade-Hermite Approximants of Vectors (ex-b5d830)

This example can be found on page 813 in [BL94a].

```
> 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: Approximants of Sine, Cosine, and Exponential (ex-15b388)

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)];
```

(continues on next page)

(continued from previous page)

```
> [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 ]
]
```

45.6 Bibliography

- [ACvHS17] Emilio Allem, Juliane Capaverde, Mark van Hoeij, and Jonas Szutkoski. Functional Decomposition using Principal Subfields. In *ISSAC '17, July 25-28, 2017, Kaiserslautern, Germany*. ACM, 2017. accepted. doi:doi.org/10.1145/3087604.3087608.
- [BL94a] 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.

Chapter 46

Algebraic Function Fields

46.1 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 *Rational Function Fields*.

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 $k(x)[y]$.

46.1.1 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.

46.2 Creation of Algebraic Function Fields and their Orders

46.2.1 Creation of Algebraic Function Fields

```
ext< K | f >: FldFunRat, RngUPolElt -> FldFun
ext< K | f >: FldFun, RngUPolElt -> FldFun
```

`FunctionField(f : parameters): RngUPolElt -> FldFun`

`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): RngMPolElt -> FldFun`

`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): [RngUPolElt] -> FldFun`

`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.

`HermitianFunctionField(p, d): RngIntElt, RngIntElt -> FldFun`

`HermitianFunctionField(q): RngIntElt -> FldFun`

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>: FldFun, [] -> FldFun`

`sub<F | s_1, ..., s_r >: FldFun, [] -> FldFun`

The subfield of the function field F containing the elements in the sequence S or the elements s_i.

`AssignNames(~F, s): FldFun, [ MonStgElt ]`

`AssignNames(~a, s): FldFunElt, [ MonStgElt ]`

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): Rng -> FldFunG`

```
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: Creating a Function Field Extension (ex-4f7d5b)

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: Creating a Relative Extension (ex-b33520)

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: Creating a Non-Simple Extension (ex-5f689c)

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: Hermitian Function Field (ex-5c6691)

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
```

46.2.2 Construction of Orders of Algebraic Function Fields

Equation orders, maximal orders, and other orders of algebraic function fields can be constructed.

For more information about the Montes algorithm which can be used to compute maximal orders, see Section *The Montes Algorithm*.

`EquationOrderFinite(F): FldFun -> RngFunOrd`

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): FldFun -> RngFunOrd`

`Al: MonStgElt`	Default:

Construct 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. If F is an extension of a rational function field over $\mathbb{Q}$ or $\mathbb{F}_q$ by a single monic integral polynomial and the parameter `Al` is set to `"Montes"` then the Montes algorithm [Sta18] will be used for the computation.

`EquationOrderInfinite(F): FldFun -> RngFunOrd`

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): FldFun -> RngFunOrd`

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): Rng, FldFun -> RngFunOrd`

The integral closure of the subring R of the function field F in itself.

`EquationOrder(O): RngFunOrd -> RngFunOrd`

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): RngFunOrd -> RngFunOrd`

```
Discriminant: Any                       Default:
Ramification: SeqEnum                   Default:
Al          : MonStgElt                 Default: "Auto"
MaximalOrder: Verbose                   Default: 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 ([BL94c], 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.

If O is an extension of a polynomial ring over $\mathbb{Q}$ or $\mathbb{F}_q$ by a single monic integral polynomial and the parameter `Al` is set to `"Montes"` then the Montes algorithm [Sta18] will be used to compute the maximal order. Otherwise `Al` may be set to `"Round2"` to avoid using the algorithms for the special cases above.

`SetOrderMaximal(O, b): RngFunOrd, BoolElt`

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>: RngFunOrd, RngUPolElt -> RngFunOrd`

```
Check: Bool                     Default: true
```

The order O with a root of f adjoined.

Example: Creating Orders of a Function Field (ex-68faea)

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);
```

(continues on next page)

(continued from previous page)

```
>> 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: Integral Closure in a Function Field (ex-4ba3aa)

```
> 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): RngFunOrd, AlgMatElt, RngElt -> RngFunOrd`

```
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). Note that this can result in a non-order being constructed which may cause errors later.

`Order(O, M): RngFunOrd, ModDed -> RngFunOrd`

```
NFBasis: BoolElt                  Default: true
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). Note that this can result in a non-order being constructed which may cause errors later. If the parameter `NFBasis` is set to `false` then the `PseudoGenerators` of the module M will be used rather than the `PseudoBasis`, however these pseudo generators must also be a pseudo basis.

`Order(O, S): RngFunOrd, [FldFunElt] -> RngFunOrd`

```
Verify : BoolElt                  Default: true
IsBasis: 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 `IsBasis` 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. Note that setting `IsBasis` to `true` can result in a non-order being constructed if the elements in the sequence are not a basis which may cause errors later.

If `IsBasis` is set to `true` to specify the basis of the resulting order rather than avoid the expense of the multiplicative closure computation, it can be checked that the result O is an order using `Order(SubOrder(O), Matrix(CoefficientRing(O), M*d), d) where d is Denominator(M) where M is BasisMatrix(O);`. If O is not an order this will cause an error.

`Simplify(O): RngFunOrd -> RngFunOrd`

Return the order O as a direct transformation of its equation order, instead of a composition of transformations.

`O1 + O2: RngFunOrd, RngFunOrd -> RngFunOrd`

The smallest common over order of $O1$ and $O2$ where $O1$ and $O2$ have the same equation order.

`O1 meet O2: RngFunOrd, RngFunOrd -> RngFunOrd`

The intersection of orders $O1$ and $O2$ which must have the same equation order.

`AsExtensionOf(O1, O2): RngFunOrd, RngFunOrd -> RngFunOrd`

Return the order $O1$ as a transformation of the order $O2$ where $O1$ and $O2$ have the same coefficient ring.

Example: Further Order Constructions (ex-b548f7)

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);
```

(continues on next page)

(continued from previous page)

```
> 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 ]
```

46.2.3 Orders and Ideals

Orders may be created using ideals of other orders. Ideals are discussed in Section *Ideals*.

MultiplicatorRing(I): RngFunOrdIdl -> RngFunOrd

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): RngFunOrd, RngFunOrdIdl -> RngFunOrd

pMaximalOrder(O, p): RngFunOrd, RngElt -> RngFunOrd

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): RngFunOrd, RngFunOrdIdl -> RngFunOrdIdl

pRadical(O, p): RngFunOrd, RngElt -> RngFunOrdIdl

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.

46.3 Related Structures

46.3.1 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): FldFun -> Cat

`Category(O): RngFunOrd -> Cat`

`Parent(F): FldFun -> Pow`

`Parent(O): RngFunOrd -> Pow`

46.3.2 Other Related Structures

More interesting related structures (than above) are listed below.

`PrimeRing(F): FldFun -> Rng`

`PrimeField(F): FldFun -> Rng`

`PrimeRing(O): RngFunOrd -> Rng`

The prime field of the function field F or the order O (prime ring of the constant field).

`ConstantField(F): FldFunG -> Rng`

`DefiningConstantField(F): FldFunG -> Rng`

The constant field k, where $F = k(x, \alpha)$.

`ExactConstantField(F): FldFunG -> Rng, Map`

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): FldFun -> Rng`

`BaseField(F): FldFun -> Rng`

`CoefficientRing(F): FldFun -> Rng`

`CoefficientField(F): FldFun -> Rng`

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): Fld, Fld -> BoolElt`

Applies to more general fields within Magma than function fields. Returns whether G is amongst the recursively defined base fields of F.

`BaseRing(O): RngFunOrd -> Rng`

`CoefficientRing(O): RngFunOrd -> Rng`

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): FldFunOrd -> Rng`

`BaseField(FF): FldFunOrd -> Rng`

`CoefficientRing(FF): FldFunOrd -> Rng`

`CoefficientField(FF): FldFunOrd -> Rng`

Given a field of fractions FF of an order O return the field of fractions of the coefficient ring of O.

`SubOrder(O): RngFunOrd -> RngFunOrd`

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): RngFunOrd -> FldFun`

The function field which O is an order of.

`FieldOfFractions(O): RngFunOrd -> FldFunOrd`

`FieldOfFractions(FF): FldFunOrd -> FldFunOrd`

`FieldOfFractions(F): FldFun -> FldFun`

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): FldFunOrd -> RngFunOrd`

Given a field of fractions FF return the order O which is the ring of integers of FF.

`RationalExtensionRepresentation(F): FldFunG -> FldFun`

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): RngFunOrd -> RngFunOrd`

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): FldFunG -> FldFunG`

The function field F expressed as an extension of its constant field.

`UnderlyingRing(F): FldFunG -> FldFunG`

`UnderlyingField(F): FldFunG -> FldFunG`

`UnderlyingRing(F, R): FldFunG, Rng -> FldFunG`

`UnderlyingField(F, R): FldFunG, Rng -> FldFunG`

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): FldFun, FldFun, FldFunElt`

`Embed(F, L, s): FldFun, FldFun, [FldFunElt]`

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.

`HasEmbedding(O1, O2): RngFunOrd, RngFunOrd -> Bool`

Print: RngIntElt	Default: 0

Return whether there is an embedding of $O1$ into $O2$ already installed. For information about the installed embedding set the `Print` parameter > 0 with higher values providing more information than lower.

`CanComputeEmbedding(O1, O2): RngFunOrd, RngFunOrd -> Bool`

Print: RngIntElt	Default: 0

Return whether an embedding of $O1$ into $O2$ can be computed. For information about the installed embedding set the `Print` parameter > 0 with higher values providing more information than lower.

`Places(F): FldFunG -> PlcFun`

The set of places of the algebraic function field F/k.

`DivisorGroup(F): FldFun -> DivFun`

The group of divisors of the algebraic function field F/k.

`DifferentialSpace(F): FldFun -> DiffFun`

The space of differentials of the algebraic function field F/k.

Example: Constant and Coefficient Fields (ex-a0477f)

```
> 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);
```

(continues on next page)

(continued from previous page)

```
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
> Places(F);
Set of places of F
> DivisorGroup(F);
Divisor group of F
```

Example: Underlying Rings of a Relative Function Field (ex-efbad3)

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): FldFun, RngIntElt -> FldFun, UserProgram`

```
Reduction: BoolElt                    Default: true
WeilRes  : Verbose                    Default: 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 [PGS00]. `Reduction` indicates whether a (possibly quite expensive) reduction step is performed at the end of the computation. It defaults to `true`.

`ConstantFieldExtension(F, E): FldFun, Rng -> FldFun, Map`

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: Extending to the Exact Constant Field (ex-01df8e)

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
```

`MonicModel(F): FldFun -> FldFun`

Given an algebraic function field F return a function field which is isomorphic to F and defined by a monic polynomial.

`Reduce(O): RngFunOrd -> RngFunOrd`

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.

`Localization(O, p): RngFunOrd, RngFunOrdIdl -> RngVal, Map`

Given an order O of an algebraic function field and a prime ideal p of O, return the localization of O at p and the map from O into the localization.

46.4 General Structure Invariants

`Characteristic(F): FldFun -> RngIntElt`

`Characteristic(O): RngFunOrd -> RngIntElt`

The characteristic of the function field F/k or one of its orders O.

`IsPerfect(F): Fld -> BoolElt`

Applies to any field in Magma. Returns whether F is perfect.

`Degree(F): FldFunG -> RngIntElt`

`Degree(F, G): FldFun, FldFun -> RngIntElt`

`Degree(O): RngFunOrd -> RngIntElt`

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): FldFunG -> RngIntElt`

`AbsoluteDegree(O): RngFunOrd -> RngIntElt`

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): FldFun -> RngUPolElt`

`DefiningPolynomial(O): RngFunOrd -> RngUPolElt`

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): FldFun -> [RngUPolElt]`

`DefiningPolynomials(O): RngFunOrd -> [RngUPolElt]`

Return the defining polynomials of the function field F or the order O as a sequence of polynomials over the coefficient ring.

`Basis(F): FldFunG -> SeqEnum[FldFunElt]`

`Basis(O): RngFunOrd -> SeqEnum[FldFunElt]`

`Basis(O, R): RngFunOrd, Rng -> SeqEnum[RngElt]`

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): RngFunOrd, RngFunOrd -> AlgMatElt, RngElt`

Return the matrix M and a denominator d which transforms elements of the order $O1$ into elements of the order $O2$.

`CoefficientIdeals(O): RngFunOrd -> [RngFunOrdIdl]`

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): RngFunOrd -> AlgMatElt`

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): RngFunOrd -> RngFunOrdElt`

A root of the defining polynomial of the order O.

`Discriminant(O): RngFunOrd -> .`

The discriminant of the order O, up to a unit in its coefficient ring.

`AbsoluteDiscriminant(O): RngFunOrd -> .`

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): FldFunG -> RngIntElt`

`DegreeOfExactConstantField(F): FldFunG -> RngIntElt`

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): FldFunG -> RngIntElt`

```
Al     : MonStgElt                    Default:
IsExact: BoolElt                      Default: false
```

The genus of the function field F/k. If F is an extension of a rational function field over $\mathbb{Q}$ or $\mathbb{F}_q$ by a single monic integral polynomial and the parameter `Al` is set to `"Montes"` then the Montes algorithm [Sta18] will be used to compute the genus.

The index $[k_0 : \text{ConstantField}(F)]$, where k_0 is the full constant field, is also returned. If it is known beforehand that k_0 is the constant field F is defined over then set the parameter `IsExact := true`.

Example: Invariants of a Relative Function Field (ex-a8d7ef)

```
> 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: Invariants of a Non-Simple Field (ex-ec93f9)

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,
```

(continues on next page)

(continued from previous page)

```
    $.1^2 + 1/$.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): FldFunG -> SeqEnum[RngIntElt]

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*.

GapNumbers(F, P): FldFunG, PlcFunElt -> SeqEnum[RngIntElt]

GapNumbers(P): PlcFunElt -> SeqEnum[RngIntElt]

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*.

SeparatingElement(F): FldFunG -> FldFunGElt

Returns a separating element of the function field F/k.

RamificationDivisor(F): FldFunG -> DivFunElt

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*.

WeierstrassPlaces(F): FldFunG -> [PlcFunElt]

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*.

`WronskianOrders(F): FldFunG -> [RngIntElt]`

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*.

`Different(O): RngFunOrd -> RngFunOrdIdl`

The different of the maximal order O.

`Index(O, S): RngFunOrd, RngFunOrd -> Any`

The index of S in O where S is a suborder of O and O and S have the same equation order.

46.5 Galois Groups

Finding Galois groups (of normal closures) of polynomials over rational function fields over $k \in \{\mathbb{Q}, \mathbb{F}_q\}$, where $\mathbb{F}_q$ denotes the finite field of characteristic p with $q = p^r$, $r \in \mathbb{Z}_{>0}$ 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 [FK14] and Sutherland [Sut15]. 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 *Galois Groups and Automorphisms*. In addition to the intrinsics described here, some of the intrinsics described in Section *Galois Groups* apply to polynomials over function fields also.

`GaloisGroup(f): RngUPolElt -> GrpPerm, [ RngElt ], GaloisData`

Prime : RngElt	Default:
NextPrime : UserProgram	Default:
ProofEffort: RngIntElt	Default: 10
Ring : GaloisData	Default:
ShortOK : BoolElt	Default: false
Old : BoolElt	Default: false
GaloisGroup: Verbose	Default: 5

Given a separable polynomial $f(t, x)$ of degree n over the rational function field $k(t)$, $k \in \{\mathbb{Q}, \mathbb{F}_q\}$, or an extension K thereof (if $k = \mathbb{F}_q$) 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 $f(t, x)$ mod $p(t)$. 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* 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. This expense can be reduced by setting `ShortOK` to `true`, in which case a descent using only the short cosets [Els14] will be considered as reliable as a descent using all cosets.

The default version employs series computations over either unramified p-adic fields ($k = \mathbb{Q}$) or finite fields ($k = \mathbb{F}_q$). 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`. If set, the `Prime` parameter must be a prime polynomial or a prime ideal when f is over an algebraic function field. In characteristic 0 a tuple of a prime polynomial and a prime integer can be given. If the `Prime` given ramifies in the extensions defined by f then an unramfied one larger than that given will be chosen.

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): FldFun -> GrpPerm, [RngElt], GaloisData
GaloisGroup(F): FldFun[FldFun[FldFunRat]] -> GrpPerm, [RngElt], GaloisData
GaloisGroup(F): FldFun[FldFin] -> GrpPerm, [RngElt], GaloisData
GaloisGroup(F): FldFun[FldFunRat] -> GrpPerm, [RngElt], GaloisData
GaloisGroup(F): FldFun[FldRat] -> GrpPerm, [RngElt], GaloisData
```

```
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`. If set, the `Prime` parameter must be a prime polynomial or a prime ideal when f is over an algebraic function field. In characteristic 0 a tuple of a prime polynomial and a prime integer can be given. If the `Prime` given ramifies in the extensions defined by f then an unramfied one larger than that given will be chosen.

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 the parameter `ShortOK` is set to `true` then a descent using only the short cosets [Els14] will be considered as reliable as a descent using all cosets.

Example: Galois Group over a Rational Function Field `(ex-0264c2)`

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];
-49 + O(191) - (39 + O(191))*t + O(t^2)
> S;
GaloisData of type Q(t)

> TransitiveGroupIdentification(G);
99 15
```

Example: Galois Groups in Positive Characteristic (ex-93f3e8)

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
```

(continues on next page)

(continued from previous page)

```
Order = 24 = 2^3 * 3
    (1, 5, 4, 7, 11, 10)(2, 6, 3, 8, 12, 9)
    (3, 10)(4, 9)(5, 12)(6, 11)
> A := Universe(r);
> AssignNames(~A,  ["t"]);
> _<w> := CoefficientRing(Universe(r));
> _<u> := CoefficientField(Parent(w));
> r;
[
    (3*u + 2)*w^5 + (u + 5)*w^4 + (5*u + 6)*w^3 + (4*u + 2)*w^2 + (5*u + 4)*w +
        3*u + 4 + ((6*u + 1)*w^5 + 6*w^4 + (3*u + 4)*w^3 + (u + 2)*w^2 + (3*u +
        3)*w + (3*u + 2))*t + ((5*u + 2)*w^4 + (2*u + 1)*w^3 + (u + 1)*w^2 +
        (3*u + 1)*w + (3*u + 3))*t^2 + ((5*u + 1)*w^5 + 3*u*w^4 + 3*u*w^3 + (3*u
        + 6)*w^2 + (4*u + 4)*w + (5*u + 4))*t^3 + O(t^4),
    (3*u + 2)*w^5 + (6*u + 2)*w^4 + (5*u + 6)*w^3 + (3*u + 5)*w^2 + (5*u + 4)*w
        + 4*u + 3 + ((6*u + 1)*w^5 + w^4 + (3*u + 4)*w^3 + (6*u + 5)*w^2 + (3*u
        + 3)*w + (4*u + 5))*t + ((2*u + 5)*w^4 + (2*u + 1)*w^3 + (6*u + 6)*w^2 +
        (3*u + 1)*w + (4*u + 4))*t^2 + ((5*u + 1)*w^5 + 4*u*w^4 + 3*u*w^3 + (4*u
        + 1)*w^2 + (4*u + 4)*w + (2*u + 3))*t^3 + O(t^4),
    (2*u + 6)*w^5 + (2*u + 3)*w^4 + (2*u + 1)*w^3 + (2*u + 1)*w^2 + (4*u + 6)*w
        + 3*u + 4 + ((4*u + 3)*w^5 + 5*w^4 + (4*u + 3)*w^3 + (4*u + 1)*w^2 + (u
        + 1)*w + (3*u + 2))*t + ((3*u + 4)*w^4 + (5*u + 6)*w^3 + (4*u + 4)*w^2 +
        (u + 5)*w + (3*u + 3))*t^2 + ((u + 3)*w^5 + 6*u*w^4 + 4*u*w^3 + (5*u +
        3)*w^2 + (6*u + 6)*w + (5*u + 4))*t^3 + O(t^4),
    (2*u + 6)*w^5 + (5*u + 4)*w^4 + (2*u + 1)*w^3 + (5*u + 6)*w^2 + (4*u + 6)*w
        + 4*u + 3 + ((4*u + 3)*w^5 + 2*w^4 + (4*u + 3)*w^3 + (3*u + 6)*w^2 + (u
        + 1)*w + (4*u + 5))*t + ((4*u + 3)*w^4 + (5*u + 6)*w^3 + (3*u + 3)*w^2 +
        (u + 5)*w + (4*u + 4))*t^2 + ((u + 3)*w^5 + u*w^4 + 4*u*w^3 + (2*u +
        4)*w^2 + (6*u + 6)*w + (2*u + 3))*t^3 + O(t^4),
    (6*u + 4)*w^5 + (3*u + 1)*w^4 + (5*u + 6)*w^3 + (6*u + 3)*w^2 + (6*u + 2)*w
        + 4*u + 3 + ((5*u + 2)*w^5 + 4*w^4 + (3*u + 4)*w^3 + (5*u + 3)*w^2 +
        (5*u + 5)*w + (4*u + 5))*t + ((u + 6)*w^4 + (2*u + 1)*w^3 + (5*u +
        5)*w^2 + (5*u + 4)*w + (4*u + 4))*t^2 + ((3*u + 2)*w^5 + 2*u*w^4 +
        3*u*w^3 + (u + 2)*w^2 + (2*u + 2)*w + (2*u + 3))*t^3 + O(t^4),
    (6*u + 4)*w^5 + (4*u + 6)*w^4 + (5*u + 6)*w^3 + (u + 4)*w^2 + (6*u + 2)*w +
        3*u + 4 + ((5*u + 2)*w^5 + 3*w^4 + (3*u + 4)*w^3 + (2*u + 4)*w^2 + (5*u
        + 5)*w + (3*u + 2))*t + ((6*u + 1)*w^4 + (2*u + 1)*w^3 + (2*u + 2)*w^2 +
        (5*u + 4)*w + (3*u + 3))*t^2 + ((3*u + 2)*w^5 + 5*u*w^4 + 3*u*w^3 + (6*u
        + 5)*w^2 + (2*u + 2)*w + (5*u + 4))*t^3 + O(t^4),
    (4*u + 5)*w^5 + (u + 5)*w^4 + (2*u + 1)*w^3 + (4*u + 2)*w^2 + (2*u + 3)*w +
        3*u + 4 + ((u + 6)*w^5 + 6*w^4 + (4*u + 3)*w^3 + (u + 2)*w^2 + (4*u +
        4)*w + (3*u + 2))*t + ((5*u + 2)*w^4 + (5*u + 6)*w^3 + (u + 1)*w^2 +
        (4*u + 6)*w + (3*u + 3))*t^2 + ((2*u + 6)*w^5 + 3*u*w^4 + 4*u*w^3 + (3*u
        + 6)*w^2 + (3*u + 3)*w + (5*u + 4))*t^3 + O(t^4),
    (4*u + 5)*w^5 + (6*u + 2)*w^4 + (2*u + 1)*w^3 + (3*u + 5)*w^2 + (2*u + 3)*w
        + 4*u + 3 + ((u + 6)*w^5 + w^4 + (4*u + 3)*w^3 + (6*u + 5)*w^2 + (4*u +
        4)*w + (4*u + 5))*t + ((2*u + 5)*w^4 + (5*u + 6)*w^3 + (6*u + 6)*w^2 +
        (4*u + 6)*w + (4*u + 4))*t^2 + ((2*u + 6)*w^5 + 4*u*w^4 + 4*u*w^3 + (4*u
        + 1)*w^2 + (3*u + 3)*w + (2*u + 3))*t^3 + O(t^4),
    (5*u + 1)*w^5 + (2*u + 3)*w^4 + (5*u + 6)*w^3 + (2*u + 1)*w^2 + (3*u + 1)*w
        + 3*u + 4 + ((3*u + 4)*w^5 + 5*w^4 + (3*u + 4)*w^3 + (4*u + 1)*w^2 +
        (6*u + 6)*w + (3*u + 2))*t + ((3*u + 4)*w^4 + (2*u + 1)*w^3 + (4*u +
        4)*w^2 + (6*u + 2)*w + (3*u + 3))*t^2 + ((6*u + 4)*w^5 + 6*u*w^4 +
        3*u*w^3 + (5*u + 3)*w^2 + (u + 1)*w + (5*u + 4))*t^3 + O(t^4),
    (5*u + 1)*w^5 + (5*u + 4)*w^4 + (5*u + 6)*w^3 + (5*u + 6)*w^2 + (3*u + 1)*w
        + 4*u + 3 + ((3*u + 4)*w^5 + 2*w^4 + (3*u + 4)*w^3 + (3*u + 6)*w^2 +
```

(continues on next page)

(continued from previous page)

```
        (6*u + 6)*w + (4*u + 5))*t + ((4*u + 3)*w^4 + (2*u + 1)*w^3 + (3*u +
        3)*w^2 + (6*u + 2)*w + (4*u + 4))*t^2 + ((6*u + 4)*w^5 + u*w^4 + 3*u*w^3
        + (2*u + 4)*w^2 + (u + 1)*w + (2*u + 3))*t^3 + O(t^4),
    (u + 3)*w^5 + (3*u + 1)*w^4 + (2*u + 1)*w^3 + (6*u + 3)*w^2 + (u + 5)*w +
        4*u + 3 + ((2*u + 5)*w^5 + 4*w^4 + (4*u + 3)*w^3 + (5*u + 3)*w^2 + (2*u
        + 2)*w + (4*u + 5))*t + ((u + 6)*w^4 + (5*u + 6)*w^3 + (5*u + 5)*w^2 +
        (2*u + 3)*w + (4*u + 4))*t^2 + ((4*u + 5)*w^5 + 2*u*w^4 + 4*u*w^3 + (u +
        2)*w^2 + (5*u + 5)*w + (2*u + 3))*t^3 + O(t^4),
    (u + 3)*w^5 + (4*u + 6)*w^4 + (2*u + 1)*w^3 + (u + 4)*w^2 + (u + 5)*w + 3*u
        + 4 + ((2*u + 5)*w^5 + 3*w^4 + (4*u + 3)*w^3 + (2*u + 4)*w^2 + (2*u +
        2)*w + (3*u + 2))*t + ((6*u + 1)*w^4 + (5*u + 6)*w^3 + (2*u + 2)*w^2 +
        (2*u + 3)*w + (3*u + 3))*t^2 + ((4*u + 5)*w^5 + 5*u*w^4 + 4*u*w^3 + (6*u
        + 5)*w^2 + (5*u + 5)*w + (5*u + 4))*t^3 + O(t^4)
]
> p;
GaloisData of type F_q(t)
> p`Prime;
t^2 + t + 4
> p`Ring;
Power series ring in t over GF(7^12)
```

GeometricGaloisGroup(f): RngUPolElt -> GrpPerm, RngUPolElt, GaloisData

Given a polynomial f over $\mathbb{Q}(t)$ return the Galois group of f as a polynomial over $\mathbb{C}(t)$ and a polynomial defining the fixed field of that group. For information about the algorithm involved see [Sut18].

HilbertIrreducibilityCurves(f): RngUPolElt -> SetEnum, SeqEnum

Given a polynomial f over $\mathbb{Q}(t)$ this intrinsic provides a finite set and a sequence of curves such that the rational points on the curves together with the finite set determine the set of exceptions to Hilbert's Irreducibility Theorem for f. For more information see [KS21].

Example: Geometric Galois Group (ex-2194f2)

We show a computation of a geometric Galois group

```
> Qt<t> := FunctionField(Rationals());
> Qtx<x> := PolynomialRing(Qt);
> f := x^9 - 3*x^7 + (-6*t - 6)*x^6 + 3*x^5 + (12*t - 6)*x^4 +
> (12*t^2 - 84*t + 11)*x^3 + (-6*t - 6)*x^2 + (-12*t^2 + 12*t + 24)*x -
> 8*t^3 - 24*t^2 - 24*t - 6;
> time GeometricGaloisGroup(f);
Permutation group acting on a set of cardinality 9
Order = 6 = 2 * 3
    (2, 9)(3, 8)(4, 7)
    (1, 9, 2)(3, 8, 6)(4, 7, 5)
x^6 + 78732
GaloisData of type Q(t)
```

and one of the exceptions and curves involved in making Hilbert's Irreducibility Theorem explicit.

```
> f := x^6 + t^6 - 1;
> HilbertIrreducibilityCurves(f);
{ -1, 1 }
[
    <Curve over Rational Field defined by
    x^2 + 6*x + 9*t^6, Permutation group acting on a set of cardinality 6
    Order = 6 = 2 * 3
```

(continues on next page)

(continued from previous page)

```
        (2, 5)(3, 4)
        (1, 5, 2)(3, 4, 6)>,
    <Curve over Rational Field defined by
    x^2 + 1728*t^12 - 3456*t^6 + 1728, Permutation group acting on a set of
    cardinality 6
    Order = 6 = 2 * 3
        (1, 3, 2, 6, 5, 4)
        (1, 5, 2)(3, 4, 6)>,
    <Curve over Rational Field defined by
    x^2 - 62208*t^30 + 311040*t^24 - 622080*t^18 + 622080*t^12 - 311040*t^6 +
        62208, Permutation group acting on a set of cardinality 6
    Order = 6 = 2 * 3
        (1, 3)(2, 4)(5, 6)
        (1, 5, 2)(3, 4, 6)>,
    <Curve over Rational Field defined by
    x^3 + 12*x^2 + 48*x - 8*t^6 + 72, Permutation group acting on a set of
    cardinality 6
    Order = 4 = 2^2
        (2, 5)(3, 4)
        (1, 6)(2, 4)(3, 5)>
]
```

46.5.1 Splitting Fields

The Galois group of a polynomial can be used to compute splitting fields of the polynomial. Magma contains intrinsics which compute a minimal degree splitting field over the coefficient field as well as a splitting field consisting of radical extensions.

`GaloisSplittingField(f): RngUPolElt -> FldFun, [FldFunElt], GrpPerm, [[FldFunElt]]`

```
Galois : Tup<GrpPerm, [RngElt], GaloisData>                    Default:
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 over $\mathbb{F}_q[t]$ this function computes the splitting field of f as a tower of fields. The various parameters 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 $\{1\}$, $\{1,2\}$, ..., $\{1,\ldots,n\}$. 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: Galois Splitting Field of a Polynomial (ex-36697a)

We start with a small example, to illustrate some of the parameters and their influence:

```
> F<t> := FunctionField(GF(5));
> P<x> := PolynomialRing(F);
> f := x^3-2*t;
> GaloisSplittingField(f);
Algebraic function field defined over Algebraic function field defined over
Univariate rational function field over GF(5) by
x^3 + 2*t by
$.1^2 + $.1*$.1 + $.1^2
[
    4*$.1,
    4*$.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
  |
  | $.1^2 + $.1*$.1 + $.1^2
  |
 $2
  |
  | x^3 + 2*t
  |
Univariate rational function field over GF(5)
Variables: t
> [x^3 : x in R];
[
    2*t,
    2*t,
    2*t
]
```

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");
Algebraic function field defined over Algebraic function field defined over
Univariate rational function field over GF(5) by
x^3 + 2*t by
K1^2 + K2*K1 + K2^2
[
    4*K2,
    4*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;
```

(continues on next page)

(continued from previous page)

```
   K<K1>
     |
     | K1^2 + K2*K1 + K2^2
     |
  $2<K2>
     |
     | x^3 + 2*t
     |
Univariate rational function field over GF(5)
Variables: t
```

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);
Algebraic function field defined over Algebraic function field defined over
Univariate rational function field over GF(5) by
x^2 + 3*t^2 by
K1^3 + 2*t
[
    4*K1,
    (3/t*K2 + 3)*K1,
    (2/t*K2 + 3)*K1
]
Symmetric group G acting on a set of cardinality 3
Order = 6 = 2 * 3
    (1, 2, 3)
    (1, 2)
[
    [
        4*K2,
        K2
    ],
    [
        4*K1,
        (3/t*K2 + 3)*K1,
        (2/t*K2 + 3)*K1
    ]
]
> (K where K := $1):Maximal;
  $1<K1>
     |
     | K1^3 + 2*t
     |
  $2<K2>
     |
     | x^2 + 3*t^2
     |
Univariate rational function field over GF(5)
Variables: t
```

And lastly we show a larger example.

```
> f := x^6 - t^14*x^4 - t^19*x^2 - t^52;
```

(continues on next page)

(continued from previous page)

```
> G, r, S := GaloisGroup(f); G;
Permutation group G acting on a set of cardinality 6
Order = 24 = 2^3 * 3
    (1, 5, 6)(2, 3, 4)
    (1, 2)(4, 5)
    (1, 5)(2, 4)
    (1, 2, 3)(4, 5, 6)
> TransitiveGroupIdentification(G);
7 6
> TransitiveGroupDescription(G);
S_4(6d) = [2^2]S(3)
> time K, R := GaloisSplittingField(f:Name := "K");
Time: 149.700
> K:Maximal;
   K<K1>
     |
     | K1^4 + (K2^2 + 4*t^14)*K1^2 + K2^4 + 4*t^14*K2^2 + 4*t^19
     |
  $2<K2>
     |
     | x^6 + 4*t^14*x^4 + 4*t^19*x^2 + 4*t^52
     |
Univariate rational function field over GF(5)
Variables: t
```

Solvability by Radicals

For a polynomial $f \in \mathbb{F}_q[t][x]$ 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 a representation.

`SolveByRadicals(f): RngUPolElt -> FldFun, [FldFunElt], [FldFunElt]`

```
Prime      : RngIntElt                                    Default: false
Name       : MonStgElt                                    Default: false
Galois     : Tup<GrpPerm, [RngElt], GaloisData>           Default:
UseZeta_p  : BoolElt                                      Default: false
MaxBound   : RngIntElt                                    Default:␣
↪Infinity()
GaloisTower: Verbose                                      Default: 3
```

For a polynomial $f \in \mathbb{Z}[t]$ 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 set, the `Prime` parameter must be a prime polynomial or a prime ideal when f is over an algebraic function field. In characteristic 0 a tuple of a prime polynomial and a prime integer can be given. If the `Prime` given ramifies in the extensions defined by f then an unramfied one larger than that given will be chosen.

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. Especially 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): FldFun, FldFunElt, RngElt -> FldFun, [FldFunElt], [FldFunElt]

Let K/F be a function 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 intrinsic will return a field L isomorphic to K such that L is a Kummer extension or an Artin–Schreier extension, i.e. the defining polynomial for L will be of the form $y^n - b$ or $y^p - y - b$ 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: Solving a Polynomial by Radicals (ex-0a9a35)

```
> F<t> := FunctionField(GF(5));
> P<x> := PolynomialRing(F);
> f := x^6 + (2*t + 3)*x^4 + 3*t*x^3 + (3*t^2 + 1)*x^2 +
> (4*t^2 + 2*t)*x + 4*t^3 + 3*t^2 + 4*t;
> K, R := SolveByRadicals(f:Name := "K.");
> K:Maximal;
   K<K.1>
     |
     | K.1^3 + 3*$.1*K.2 + 4*$.1
     |
  $2<K.2>
     |
     | K.2^2 + 2*$.1^2 + 1
     |
  $3<K.3>
     |
     | K.3^2 + $.1
     |
Univariate rational function field over GF(5^2)
Variables: $.1
> [ 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 $x^n - a$ or $x^p - x - a$ in characteristic p.

46.6 Subfields

For finite extensions L of $\mathbb{Q}(t)$, $\mathbb{F}_q(t)$ or extensions K of $\mathbb{F}_q(t)$ Magma can compute all fields between L and $\mathbb{Q}(t)$, $\mathbb{F}_q(t)$, or K. It should be noted that the computation of subfields does not depend on the Galois groups. The implementation over $\mathbb{Q}(t)$ uses the algorithm of Klüners [Klu02]. The implementation over $\mathbb{F}_q(t)$ and extensions thereof follows the newer ideas of Klüners and van Hoeij [vHKN11].

Subfields(F): FldFun -> SeqEnum[FldFun]

All algebraic function fields G with $k(x) \subset G \subseteq F$ or $K \subset G \subseteq F$ where K is an extension of $\mathbb{F}_q(x)$ and the coefficient field of F.

Example: Subfields of a Function Field (ex-aeb7db)

A subfield computation is shown below.

```
> k<x>:= FunctionField(Rationals());
> R<y>:= PolynomialRing(k);
```

(continues on next page)

(continued from previous page)

```
> 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>
]
```

46.7 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 [Hes]. Here the constant field of K can, in fact, be any exact perfect field in Magma with a few provisos.

46.7.1 Automorphisms over the Base Field

`Automorphisms(K, k): FldFun, FldFunG -> [Map]`

Computes all $\mathbb{Q}(t)$ automorphisms of the absolute finite extension K that fix k. The field k has to be $\mathbb{Q}(t)$ for this function.

`AutomorphismGroup(K, k): FldFun, FldFunG -> GrpFP, Map`

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 $\mathbb{Q}(t)$.

Example: Automorphisms of a Degree 7 Extension (ex-796d6f)

We define an extension of degree 7 over $\mathbb{Q}(t)$ 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,
```

(continues on next page)

(continued from previous page)

```
    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): FldFun, FldFun -> BoolElt, Map`

Given two absolute finite extensions K and L of $\mathbb{Q}(t)$, decide if L is an extension of K. If this is the case, return an embedding map from K into L.

`IsIsomorphicOverQt(K, L): FldFun, FldFun -> BoolElt, Map`

Given two absolute finite extensions K and L of $\mathbb{Q}(t)$, decide if L is $\mathbb{Q}(t)$-isomorphic to K. If this is the case, return a map from K onto L.

Example: Subfields and Isomorphism Testing (ex-40fb2d)

`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
```

46.7.2 General Automorphisms

`Isomorphisms(K, E): FldFunG, FldFunG -> [Map]`

```
BaseMorphism: Map                          Default: false
Bound       : RngIntElt                    Default: Infinity()
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): FldFunG, FldFunG -> BoolElt, Map`

```
BaseMorphism: Map                          Default: false
Strategy    : MonStgElt                    Default: "None"
```

As above, except the function only computes a single isomorphism if one exists.

`Automorphisms(K): FldFunG -> [Map]`

```
BaseMorphism: Map                          Default: false
Bound       : RngIntElt                    Default: Infinity()
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 constructors 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): FldFunG, FldFunG, PlcFunElt, PlcFunElt -> [Map]`

`Automorphisms(K, p1, p2): FldFunG, PlcFunElt, PlcFunElt -> [Map]`

```
Bound: RngIntElt                       Default: Infinity()
```

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): FldFunG -> GrpFP, Map`

```
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): FldFunG, Map -> Grp, Map, SeqEnum`

`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`.

46.7.3 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): Map -> Bool`

Returns `true`, if the map is a field or function field morphism; `false` otherwise.

`FieldMorphism(f): Map -> Map`

Converts a homomorphism between fields into a field morphism.

`IdentityFieldMorphism(F): Fld -> Map`

Returns the identity automorphism of field F as a field morphism.

`IsIdentity(f): Map -> BoolElt`

Returns `true` if f is the identity morphism; `false` otherwise.

`Equality(f, g): Map, Map -> Bool`

Returns `true`, if the two maps are both field morphisms or function field morphisms and are equal; `false` otherwise.

`HasInverse(f): Map -> MonStgElt, Map`

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): Map, Map -> Map`

The composition of the field morphisms f and g.

Example: Isomorphisms of Function Fields (ex-9d9eb5)

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 $\mathbb{Q}(i)$ $[i^2 = -1]$ and a Galois twist of it. The fields are not isomorphic over $\mathbb{Q}(i)$ but they are over $\mathbb{Q}$:

```
> 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 $PGL_2(\mathbb{F}_7)$. 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);
```

(continues on next page)

(continued from previous page)

```
> 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 $\mathbb{F}_5$ 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
```

46.8 Global Function Fields

In this section F/k denotes a global function field.

46.8.1 Functions relative to the Exact Constant Field

`NumberOfPlacesOfDegreeOverExactConstantField(F, m): FldFun, RngIntElt -> RngIntElt`

`NumberOfPlacesDegECF(F, m): FldFun, RngIntElt -> RngIntElt`

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): FldFunG -> RngIntElt`

`NumberOfPlacesOfDegreeOneECF(F): FldFunG -> RngIntElt`

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): FldFunG, RngIntElt -> RngIntElt`

`NumberOfPlacesOfDegreeOneECF(F, m): FldFun, RngIntElt -> RngIntElt`

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): FldFunG -> RngIntElt`

`SerreBound(F, m): FldFunG, RngIntElt -> RngIntElt`

`SerreBound(q, g): RngIntElt, RngIntElt -> RngIntElt`

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): FldFunG -> RngIntElt`

`IharaBound(F, m): FldFunG, RngIntElt -> RngIntElt`

`IharaBound(q, g): RngIntElt, RngIntElt -> RngIntElt`

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): FldFunG -> RngIntElt`

`NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(F): FldFunG -> RngIntElt`

`NumberOfPlacesOfDegreeOneECFBound(F, m): FldFun, RngIntElt -> RngIntElt`

`NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(F, m): FldFunG, RngIntElt -> RngIntElt`

`NumberOfPlacesOfDegreeOneECFBound(q, g): RngIntElt, RngIntElt -> RngIntElt`

`NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(q, g): RngIntElt, RngIntElt -> RngIntElt`

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): FldFunG -> RngUPolElt`

The L-polynomial of the global function field F/k (with respect to the exact constant field).

`LPolynomial(F, m): FldFunG, RngIntElt -> RngUPolElt`

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): FldFunG -> FldFunRatUElt`

The Zeta function of the global function field F/k (with respect to the exact constant field).

`ZetaFunction(F, m): FldFunG, RngIntElt -> FldFunRatUElt`

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).

46.8.2 Functions Relative to the Constant Field

`Places(F, m): FldFunG, RngIntElt -> SeqEnum[PlcFunElt]`

A sequence containing the places of degree m of the global function field F/k.

`HasPlace(F, m): FldFunG, RngIntElt -> BoolElt, PlcFunElt`

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): FldFunG, RngIntElt -> BoolElt, PlcFunElt`

Returns `true` and a random place of degree m in the global function field (`false` if there are none).

`RandomPlace(F, m): FldFunG, RngIntElt -> PlcFunElt`

Returns a random place of degree m in the global function field or throws an error if there is none.

Example: Invariants of a Global Function Field (ex-8f580e)

```
> 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: Invariants of a Global Relative Field (ex-9f3184)

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
```

46.8.3 Functions related to Class Group

`UnitRank(O): RngFunOrd -> RngIntElt`

Given a maximal 'finite' order O in a global function field, return the unit rank of O.

UnitGroup(O): RngFunOrd -> GrpAb, Map

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*.

Regulator(O): RngFunOrd -> RngIntElt

The regulator of the unit group of the 'finite' maximal order O.

PrincipalIdealMap(O): RngFunOrd -> Map

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: Unit Group of an Equation Order (ex-1591a6)

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): FldFunG -> GrpAb, Map, Map

```
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*.

ClassGroup(O): RngFunOrd -> GrpAb, Map, Map

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): RngFunOrd -> Map, Map, Map

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): FldFunG -> SeqEnum

```
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*.

ClassGroupAbelianInvariants(O): RngFunOrd -> SeqEnum

Computes a sequence of integers containing the Abelian invariants of the ideal class group of the 'finite' maximal order O.

ClassNumber(F): FldFunG -> RngIntElt

The order of the group of divisor classes of degree zero of F/k.

ClassNumber(O): RngFunOrd -> RngIntElt

The order of the ideal class group of the 'finite' maximal order O.

Example: Class Groups of Relative Fields (ex-dd2e5e)

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): FldFunG -> GrpAb, Map

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* and *IsGlobalUnitWithPreimage*.

ClassGroupPRank(F): FldFunG -> RngIntElt

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*.

HasseWittInvariant(F): FldFunG -> RngIntElt

Return the Hasse–Witt invariant of F/k. See *HasseWittInvariant* for a detailed description.

IndependentUnits(O): RngFunOrd -> SeqEnum[RngFunOrdElt]

A sequence of independent units of the 'finite' maximal order O.

FundamentalUnits(O): RngFunOrd -> SeqEnum[RngFunOrdElt]

A sequence of fundamental units of the 'finite' maximal order O.

Example: Basis and Fundamental Units of a Maximal Order (ex-065c59)

```
> 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
```

46.9 Structure Predicates

IsField(R): FldFun -> BoolElt

IsEuclideanDomain(R): FldFun -> BoolElt

IsField(O): RngFunOrd -> BoolElt

IsPID(R): FldFun -> BoolElt

IsUFD(R): FldFun -> BoolElt

IsDivisionRing(R): FldFun -> BoolElt

IsEuclideanRing(R): FldFun -> BoolElt

IsDivisionRing(O): RngFunOrd -> BoolElt

IsPrincipalIdealRing(R): FldFun -> BoolElt

IsDomain(R): FldFun -> BoolElt

IsDomain(O): RngFunOrd -> BoolElt

F **eq** G: FldFunG, FldFunG -> BoolElt

F **ne** G: FldFunG, FldFunG -> BoolElt

O1 **eq** O2: RngFunOrd, Rng -> BoolElt

O1 **ne** O2: RngFunOrd, Rng -> BoolElt

O1 **subset** O2: RngFunOrd, RngFunOrd -> BoolElt

Return whether $O1$ is a subset of $O2$.

IsGlobal(F): FldFunG -> BoolElt

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): FldFunG -> BoolElt

Return **true** if the function field F is isomorphic to a rational function field, (i.e. F is only trivially algebraic).

`IsFiniteOrder(O): RngFunOrd -> BoolElt`

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): RngFunOrd -> BoolElt`

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): RngFunOrd -> BoolElt`

Return `false` if the order O is an extension of another order, otherwise `true`.

`IsMaximal(O): RngFunOrd -> BoolElt`

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): RngFunOrd -> BoolElt`

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): RngFunOrd -> BoolElt`

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): RngFunOrd -> BoolElt`

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): RngFunOrd -> BoolElt`

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): FldFun -> BoolElt, FldFunElt`

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 multiplicative 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): FldFun -> BoolElt, FldFunElt`

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.

46.10 Homomorphisms

`hom<F -> R | g>: FldFun, Rng, RngElt -> Map`

`hom<F -> R | cf, g>: FldFun, Rng, Map, RngElt -> Map`

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 >: RngFunOrd, Rng, RngElt -> Map`

`hom< O -> R | cf, g >: RngFunOrd, Rng, Map, RngElt -> Map`

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): Map -> BoolElt`

Return whether the vector space homomorphism m is a homomorphism of rings.

Example: Homomorphisms of Relative Function Fields (ex-1107ee)

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`$_n$` >: RngFunOrd, Rng, RngElt, ..., RngElt -> Map`

`hom< O -> R | m, b`$_1$`, ..., b`$_n$` >: RngFunOrd, Rng, Map, RngElt, ..., RngElt -> Map`

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.

46.11 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.

46.11.1 Creation of Elements

`F . 1: FldFun -> FldFunElt`

`F . 2: FldFun -> FldFunElt`

(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): FldFun, RngIntElt -> FldFunElt`

Given a function field F, return the i-th generator, i.e. return the element `F.1` or `F.2` of F.

`O . i: RngFunOrd, RngIntElt -> FldFunOrdElt`

`FF . i: FldFunOrd, RngIntElt -> FldFunOrdElt`

Return the ith basis element of the order O or its field of fractions FF.

`F ! a: FldFun, . -> FldFunElt`

`elt<F | a>: FldFun, . -> FldFunElt`

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 $a_0 + a_1\alpha + \cdots + a_{n-1}\alpha^{n-1}$ 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: RngFunOrd, . -> RngFunOrdElt`

`elt<O | a>: RngFunOrd, . -> RngFunOrdElt`

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 $a_1\omega_1 + a_2\omega_2 + \cdots + a_n\omega_n$ 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: FldFunOrd, Any -> FldFunOrdElt`

`elt<FF | a>: FldFunOrd, Any -> FldFunOrdElt`

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 | a0, a1, ..., an-1>: FldFun, RngElt, ..., RngElt -> FldFunElt`

Create the element $a_0 + a_1\alpha + \cdots + a_{n-1}\alpha^{n-1}$ 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 | a1, a2, ..., an>: RngFunOrd, RngElt, ..., RngElt -> RngFunOrdElt`

`elt< FF | a1, a2, ..., an>: FldFunOrd, RngElt, ..., RngElt -> FldFunOrdElt`

Create the element $a_1\omega_1 + a_2\omega_2 + \cdots + a_n\omega_n$ 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): FldFun -> FldFunElt`

`One(O): RngFunOrd -> RngFunOrdElt`

`Identity(F): FldFun -> FldFunElt`

`Identity(O): RngFunOrd -> RngFunOrdElt`

`Zero(F): FldFun -> FldFunElt`

`Zero(O): RngFunOrd -> RngFunOrdElt`

`Representative(F): FldFun -> FldFunElt`

`Representative(O): RngFunOrd -> RngFunOrdElt`

These generic functions (cf. Chapter Introduction to Rings) create 1, 1, 0, and 0 respectively in the function field F or order O.

`Random(F, m): FldFunG, RngIntElt -> FldFunElt`

`Random(O, m): RngFunOrd, RngIntElt -> RngFunOrdElt`

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.

46.11.2 Parent and Category

`Parent(a): FldFunElt -> FldFun`

`Parent(a): RngFunOrdElt -> RngFunOrd`

`Category(a): FldFunElt -> Cat`

`Category(a): RngFunOrdElt -> Cat`

46.11.3 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): FldFunElt -> SeqEnum[FldElt]`

`ElementToSequence(a): RngFunOrdElt -> SeqEnum[RngElt]`

`ElementToSequence(a): FldFunOrdElt -> SeqEnum[RngElt]`

`Eltseq(a): FldFunElt -> SeqEnum[FldElt]`

`Eltseq(a): RngFunOrdElt -> SeqEnum[RngElt]`

`Eltseq(a): FldFunOrdElt -> SeqEnum[RngElt]`

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 $a = a_1\omega_1 + a_2\omega_2 + \cdots + a_n\omega_n$ where $\omega_1, \omega_2, \ldots, \omega_n$ is a basis of the parent of a.

`Eltseq(a, R): FldFunElt, FldFunG -> [FldFunGElt]`

A sequence of coefficients of the function field element a in the coefficient field R.

`Flat(a): FldFunElt -> [FldFunGElt]`

A sequence of coefficients of the function field element a in the bottom coefficient field of the parent of a.

`F ! [ a0, a1, ..., an-1 ]: FldFun, SeqEnum -> FldFunElt`

The element $a = a_0 + a_1\alpha + \cdots + a_{n-1}\alpha^{n-1}$ 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 ! [ a1, a2, ..., an ]: RngFunOrd, SeqEnum -> RngFunOrdElt`

The element $a = a_1\omega_1 + a_2\omega_2 + \cdots + a_n\omega_n$ 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: Creating Function Field Elements (ex-27bc08)

```
> 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;
```

(continues on next page)

(continued from previous page)

```
alpha
> b := x + alpha + 1/x*alpha^2;
> b;
1/x*alpha^2 + alpha + x
> b eq F ! [x, 1, 1/x];
true
```

46.11.4 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: FldFunElt -> FldFunElt`

`+ a: RngFunOrdElt -> RngFunOrdElt`

`+ a: FldFunOrdElt -> FldFunOrdElt`

`- a: FldFunElt -> FldFunElt`

`- a: RngFunOrdElt -> RngFunOrdElt`

`- a: FldFunOrdElt -> FldFunOrdElt`

`a + b: FldFunElt, FldFunElt -> FldFunElt`

`a + b: RngFunOrdElt, RngFunOrdElt -> RngFunOrdElt`

`a + b: FldFunOrdElt, FldFunOrdElt -> FldFunOrdElt`

`a - b: FldFunElt, FldFunElt -> FldFunElt`

`a - b: RngFunOrdElt, RngFunOrdElt -> RngFunOrdElt`

`a - b: FldFunOrdElt, FldFunOrdElt -> FldFunOrdElt`

`a * b: FldFunElt, FldFunElt -> FldFunElt`

`a * b: RngFunOrdElt, RngFunOrdElt -> RngFunOrdElt`

`a * b: FldFunOrdElt, FldFunOrdElt -> FldFunOrdElt`

`a div b: FldFunElt, FldFunElt -> FldFunElt`

`a div b: FldFunElt, RngElt -> FldFunElt`

`a div b: RngFunOrdElt, RngFunOrdElt -> RngFunOrdElt`

`a div b: RngFunOrdElt, RngElt -> RngFunOrdElt`

`a / b: FldFunElt, FldFunElt -> FldFunElt`

`a / b: RngFunOrdElt, RngFunOrdElt -> FldFunGElt`

`a / b: FldFunOrdElt, FldFunOrdElt -> FldFunOrdElt`

`a ^ k: FldFunElt, RngIntElt -> FldFunElt`

`a ^ k: RngFunOrdElt, RngIntElt -> RngFunOrdElt`

`a ^ k: FldFunOrdElt, RngIntElt -> FldFunOrdElt`

`Modexp(a, k, m): RngFunOrdElt, RngIntElt, RngUPolElt -> RngFunOrdElt`

`Modexp(a, k, m): RngFunOrdElt, RngIntElt, RngValElt -> RngFunOrdElt`

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: RngFunOrdElt, RngFunOrdIdl -> RngFunOrdElt`

Return the element a belonging to the order O as an element of O/I.

`Modinv(a, m): RngFunOrdElt, RngFunOrdIdl -> RngFunOrdElt`

`Modinv(a, m): RngFunOrdElt, RngUPolElt -> RngFunOrdElt`

`Modinv(a, m): RngFunOrdElt, RngValElt -> RngFunOrdElt`

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.

46.11.5 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: FldFunElt, FldFunElt -> BoolElt

a **eq** b: RngFunOrdElt, RngFunOrdElt -> BoolElt

a **eq** b: FldFunOrdElt, FldFunOrdElt -> BoolElt

a **ne** b: FldFunElt, FldFunElt -> BoolElt

a **ne** b: RngFunOrdElt, RngFunOrdElt -> BoolElt

a **ne** b: FldFunOrdElt, FldFunOrdElt -> BoolElt

a **in** F: RngElt, FldFun -> BoolElt

a **in** O: RngElt, RngFunOrd -> BoolElt

a **in** FF: RngElt, FldFunOrd -> BoolElt

a **notin** F: RngElt, FldFun -> BoolElt

a **notin** O: RngElt, RngFunOrd -> BoolElt

a **notin** FF: RngElt, FldFunOrd -> BoolElt

46.11.6 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): FldFunElt, FldFunElt -> BoolElt, FldFunElt

IsDivisibleBy(a, b): RngFunOrdElt, RngFunOrdElt -> BoolElt, RngFunOrdElt

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): FldFunElt -> BoolElt

IsZero(a): RngFunOrdElt -> BoolElt

IsZero(a): FldFunOrdElt -> BoolElt

IsOne(a): FldFunElt -> BoolElt

IsOne(a): RngFunOrdElt -> BoolElt

IsOne(a): FldFunOrdElt -> BoolElt

IsMinusOne(a): FldFunElt -> BoolElt

IsMinusOne(a): RngFunOrdElt -> BoolElt

IsMinusOne(a): FldFunOrdElt -> BoolElt

IsNilpotent(a): FldFunElt -> BoolElt

IsNilpotent(a): RngFunOrdElt -> BoolElt

IsIdempotent(a): FldFunElt -> BoolElt

IsIdempotent(a): RngFunOrdElt -> BoolElt

IsUnit(a): FldFunElt -> BoolElt

IsUnit(a): RngFunOrdElt -> BoolElt

IsUnit(a): FldFunOrdElt -> BoolElt

IsZeroDivisor(a): FldFunElt -> BoolElt

IsZeroDivisor(a): RngFunOrdElt -> BoolElt

`IsRegular(a): FldFunElt -> BoolElt`

`IsRegular(a): RngFunOrdElt -> BoolElt`

`IsIrreducible(a): FldFunElt -> BoolElt`

`IsIrreducible(a): RngFunOrdElt -> BoolElt`

`IsPrime(a): FldFunElt -> BoolElt`

`IsPrime(a): RngFunOrdElt -> BoolElt`

`IsSeparating(a): FldFunGElt -> BoolElt`

Returns `true` if the function field element a is a separating element (has a non zero differential).

`IsConstant(a): FldFunGElt -> BoolElt, RngElt`

`IsConstant(a): RngFunOrdElt -> BoolElt, RngElt`

Whether the algebraic function a is constant; if so it is returned as an element of the exact constant field.

`IsGlobalUnit(a): FldFunElt -> BoolElt`

Whether the function field element a is a global unit, i.e. a constant (equivalent to `IsConstant`)

`IsGlobalUnitWithPreimage(a): FldFunElt -> BoolElt, GrpAbElt`

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): RngFunOrdElt -> BoolElt, GrpAbElt`

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.

46.11.7 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): FldFunGElt -> FldFunGElt`

`Trace(a): RngFunOrdElt -> RngElt`

`Norm(a): FldFunGElt -> FldFunGElt`

`Norm(a): RngFunOrdElt -> RngElt`

`MinimalPolynomial(a): FldFunElt -> RngUPolElt`

`MinimalPolynomial(a): RngFunOrdElt -> RngUPolElt`

`MinimalPolynomial(a): FldFunOrdElt -> RngUPolElt`

`CharacteristicPolynomial(a): FldFunElt -> RngUPolElt`

`CharacteristicPolynomial(a): RngFunOrdElt -> RngUPolElt`

`RepresentationMatrix(a): FldFunGElt -> AlgMatElt`

`RepresentationMatrix(a): RngFunOrdElt -> AlgMatElt`

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): FldFunElt, Rng -> RngElt`

`Trace(a, R): RngFunOrdElt, Rng -> RngElt`

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): FldFunElt, Rng -> RngElt`

`Norm(a, R): RngFunOrdElt, Rng -> RngElt`

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): FldFunElt, Rng -> RngUPolElt`

`CharacteristicPolynomial(a, R): RngFunOrdElt, Rng -> RngUPolElt`

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): FldFunElt, Rng -> RngUPolElt`

`MinimalPolynomial(a, R): RngFunOrdElt, Rng -> RngUPolElt`

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): FldFunElt -> RngUPolElt`

The minimal polynomial of the function field element a over the rational function field.

`RepresentationMatrix(a, R): FldFunGElt, Rng -> AlgMatElt`

`RepresentationMatrix(a, R): RngFunOrdElt, Rng -> AlgMatElt`

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: Norm and Trace of Elements (ex-2d04b2)

```
> 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
```

46.11.8 Functions related to Orders and Integrality

`IntegralSplit(a, O): FldFunElt, RngFunOrd -> RngFunOrdElt, RngElt`

`IntegralSplit(a, O): RngFunOrdElt, RngFunOrd -> RngFunOrdElt, RngElt`

`IntegralSplit(a, O): FldFunOrdElt, RngFunOrd -> RngFunOrdElt, RngElt`

Split the element function field or order element a into a numerator and denominator with respect to the order O.

`Numerator(a, O): FldFunElt, RngFunOrd -> RngFunOrdElt`

The numerator of the function field element a with respect to the order O.

`Numerator(a): FldFunOrdElt -> RngFunOrdElt`

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): FldFunOrdElt, RngFunOrd -> RngElt`

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): FldFunElt, RngFunOrd -> RngElt`

The denominator of the function field element a with respect to the order O.

`Denominator(a): FldFunOrdElt -> RngElt`

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): FldFunOrdElt, RngFunOrd -> RngElt`

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): FldFunElt, RngFunOrd -> RngElt, RngElt`

`Minimum(a, O): FldFunElt, RngFunOrd -> RngElt, RngElt`

`Minimum(a, O): RngFunOrdElt, RngFunOrd -> RngElt, RngElt`

`Min(a, O): RngFunOrdElt, RngFunOrd -> RngElt, RngElt`

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).

46.11.9 Functions related to Places and Divisors

`Evaluate(a, P): FldFunElt, PlcFunElt -> RngElt`

`Evaluate(a, P): RngFunOrdElt, PlcFunElt -> RngElt`

Evaluate the algebraic function a at the place P. If it is not defined at P, infinity is returned.

`Lift(a, P): RngElt, PlcFunElt -> FldFunElt`

Lift the element a of the residue class field of the place P (including infinity) to an algebraic function.

`Valuation(a, P): FldFunElt, PlcFunElt -> RngIntElt`

`Valuation(a, P): RngFunOrdElt, PlcFunElt -> RngIntElt`

The valuation of the function field or order element a at the place P.

`Expand(a, P): FldFunGElt, PlcFunElt -> RngSerElt, FldFunGElt`

```
RelPrec: RngIntElt                    Default: 10
AbsPrec: RngIntElt                    Default:
```

Expand the algebraic function a to a series of given precision at the place P and return also the local parameter.

`Development(a, P): FldFunGElt, PlcFunElt -> RngSerElt`

```
RelPrec: RngIntElt                    Default: 10
AbsPrec: RngIntElt                    Default:
```

The algebraic function a expressed as a combination of powers of the local parameter of the place P.

`Divisor(a): FldFunGElt -> DivFunElt`

`Divisor(a): RngFunOrdElt -> DivFunElt`

`PrincipalDivisor(a): FldFunGElt -> DivFunElt`

`PrincipalDivisor(a): RngFunOrdElt -> DivFunElt`

The (principal) divisor (a) of the function field or order element a.

`Zeros(a): FldFunGElt -> [PlcFunElt]`

`Zeros(a): RngFunOrdElt -> [PlcFunElt]`

`Zeroes(a): FldFunGElt -> [PlcFunElt]`

`Zeroes(a): RngFunOrdElt -> [PlcFunElt]`

A sequence containing the zeros of the algebraic function a.

`Zeros(F, a): FldFunG, FldFunGElt -> [PlcFunElt]`

`Zeroes(F, a): FldFunG, FldFunGElt -> [PlcFunElt]`

The zeros of the function field element a in the function field F.

`Poles(a): FldFunGElt -> SeqEnum[PlcFunElt]`

`Poles(a): RngFunOrdElt -> SeqEnum[PlcFunElt]`

A sequence containing the poles of the algebraic function a.

`Poles(F, a): FldFun, FldFunGElt -> [PlcFunElt]`

A sequence containing the poles of the function field element a in the function field F.

`Degree(a): FldFunElt -> RngIntElt`

`Degree(a): RngFunOrdElt -> RngIntElt`

The degree of the algebraic function a, being defined as the degree of the pole (or zero) divisor of a.

`CommonZeros(L): [FldFunGElt] -> [PlcFunElt]`

Return the common zeros of the function field elements in the sequence L.

`CommonZeros(F, L): FldFunG, SeqEnum[ FldFunGElt ] -> SeqEnum[ PlcFunElt ]`

Return the common zeros in the function field F of the function field elements in the sequence L.

Example: Minimal Polynomial and Representation Matrix (ex-0c032c)

```
> 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
```

(continues on next page)

(continued from previous page)

```
> Valuation(a, P);
0
> Valuation(a-c, P);
1
```

`Module(L, R): SeqEnum[ FldFunGElt ], Rng -> Mod, Map, SeqEnum[ ModElt ]`

```
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): SeqEnum[ FldFunElt ], Rng -> ModTupRng`

`Relations(L, R, m): SeqEnum[ FldFunElt ], Rng, RngIntElt -> ModTupRng`

The module of R-linear relations between the function field elements of the sequence L. The integer 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 $M := \{ \sum_{i=1}^{m} v_i a_i \,|\, v = (v_i)_i \in V \}$. 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): RngUPolElt, DivFunElt -> SeqEnum[ FldFunElt ]`

Compute the roots of the polynomial f which lie in the Riemann-Roch space of the divisor D.

Example: Module and Relations over Relative Fields (ex-7f09fd)

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
```

46.11.10 Other Operations on Elements

`ProductRepresentation(a): FldFunGElt -> [FldFunGElt], [RngIntElt]`

`ProductRepresentation(a): RngFunOrdElt -> [RngElt], [RngIntElt]`

Return a product representation for the function field or order element a.

`ProductRepresentation(Q, S): [FldFunGElt], [RngIntElt] -> FldFunGElt`

`PowerProduct(Q, S): [FldFunGElt], [RngIntElt] -> FldFunGElt`

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): FldFunGElt -> RngElt`

`RationalFunction(a, R): FldFunGElt, Rng -> RngElt`

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): FldFunGElt, FldFunGElt -> FldFunGElt`

The first differentiation resp. derivative of the function field element a with respect to the separating element x.

`Differentiation(x, n, a): FldFunGElt, RngIntElt, FldFunGElt -> FldFunGElt`

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): FldFunGElt, RngIntElt, FldFunGElt -> SeqEnum`

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): FldFunGElt, RngIntElt, FldFunGElt -> SeqEnum`

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): RngFunOrdElt -> RngFunOrdElt`

The different of the element a of an order of an algebraic function field.

`RationalReconstruction(e, f): FldFunElt, RngUPolElt -> BoolElt, FldFunElt`

For an element e of 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): RngFunOrdElt -> RngIntElt`

`CoefficientHeight(a): FldFunElt -> RngIntElt`

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): RngFunOrdElt -> RngIntElt`

`CoefficientLength(a): FldFunElt -> RngIntElt`

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: Product Representation of Elements (ex-d8f7df)

```
> 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 ], [
```

(continues on next page)

(continued from previous page)

```
    (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 ] ]
```

46.12 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 $[F : k(x, \alpha_1, \ldots, \alpha_r)]$.

46.12.1 Creation of Ideals

`ideal< O | a`$_1$`, a`$_2$`, ... , a`$_m$` >: RngFunOrd, RngElt, ..., RngElt -> RngFunOrdIdl`

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 >: RngFunOrd, AlgMatElt, RngElt -> RngFunOrdIdl`

`ideal< O | T, d >: RngFunOrd, ModDed, RngElt -> RngFunOrdIdl`

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 >: RngFunOrd, AlgMatElt, [RngFunOrdIdl] -> RngFunOrdIdl`

`ideal< O | T, I`$_1$`, ..., I`$_n$` >: RngFunOrd, AlgMatElt, RngFunOrdIdl, ..., RngFunOrdIdl -> RngFunOrdIdl`

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: RngElt, RngFunOrd -> RngFunOrdIdl`

`O * x: RngFunOrd, RngElt -> RngFunOrdIdl`

Create the ideal $x * O$ where x is coercible into the function field of the order O.

`Ideal(P): PlcFunElt -> RngFunOrdIdl`

Create a prime ideal corresponding to the place P.

`Ideals(D): DivFunElt -> RngFunOrdIdl, RngFunOrdIdl`

Create two ideals of the 'finite' and 'infinite' maximal order respectively corresponding to the divisor D.

`O !! I: RngFunOrd, RngFunOrdIdl -> RngFunOrdIdl`

Return the ideal I as an ideal of the order O.

46.12.2 Parent and Category

`Parent(I): RngFunOrdIdl -> PowIdl`

`Category(I): RngFunOrdIdl -> Cat`

46.12.3 Arithmetic Operators

`I + J: RngFunOrdIdl, RngFunOrdIdl -> RngFunOrdIdl`

`I * J: RngFunOrdIdl, RngFunOrdIdl -> RngFunOrdIdl`

`I / J: RngFunOrdIdl, RngFunOrdIdl -> RngFunOrdIdl`

`I ^ k: RngFunOrdIdl, RngIntElt -> RngFunOrdIdl`

The denominator ideal J is required to be invertible. The ideal I is required to be invertible for negative k.

`c * I: RngElt, RngFunOrdIdl -> RngFunOrdIdl`

`I * c: RngFunOrdIdl, RngElt -> RngFunOrdIdl`

`I / c: RngFunOrdIdl, RngElt -> RngFunOrdIdl`

`c / I: RngElt, RngFunOrdIdl -> RngFunOrdIdl`

The principal ideal generated by the ring element c divided by the ideal I.

`IdealQuotient(I, J): RngFunOrdIdl, RngFunOrdIdl -> RngFunOrdIdl`

`ColonIdeal(I, J): RngFunOrdIdl, RngFunOrdIdl -> RngFunOrdIdl`

The colon ideal $[I : J]$ of elements which multiply all elements of the ideal J into the ideal I.

`ChineseRemainderTheorem(I1, I2, e1, e2): RngFunOrdIdl, RngFunOrdIdl, RngFunOrdElt, RngFunOrdElt -> RngFunOrdElt`

`CRT(I1, I2, e1, e2): RngFunOrdIdl, RngFunOrdIdl, RngFunOrdElt, RngFunOrdElt -> RngOrdElt`

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.

46.12.4 Roots of Ideals

`IsPower(I, n): RngFunOrdIdl, RngIntElt -> BoolElt, RngFunOrdIdl`

Return whether the ideal I has an nth root and if so return an nth root.

`Root(I, n): RngFunOrdIdl, RngIntElt -> RngFunOrdIdl`

Return the nth root of the ideal I.

`IsSquare(I): RngFunOrdIdl -> BoolElt, RngFunOrdIdl`

Return whether the ideal I is a square and if so return a square root.

`SquareRoot(I): RngFunOrdIdl -> RngFunOrdIdl`

`Sqrt(I): RngFunOrdIdl -> RngFunOrdIdl`

Return a square root of the ideal I.

Example: Testing Whether an Ideal Is a Square (ex-eaf22a)

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
```

(continues on next page)

(continued from previous page)

```
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
```

46.12.5 Equality and Membership

`I eq J: RngFunOrdIdl, RngFunOrdIdl -> BoolElt`

`I ne J: RngFunOrdIdl, RngFunOrdIdl -> BoolElt`

`I in S: RngFunOrdIdl, PowIdl -> BoolElt`

`I notin S: RngFunOrdIdl, PowIdl -> BoolElt`

46.12.6 Predicates on Ideals

`IsZero(I): RngFunOrdIdl -> BoolElt`

Returns `true` if and only if the ideal I is the zero ideal of the order O.

`IsOne(I): RngFunOrdIdl -> BoolElt`

Returns `true` if and only if the ideal I is the identity ideal of the order O, i.e. $I = O$.

`IsIntegral(I): RngFunOrdIdl -> BoolElt`

Returns `true` if and only if the ideal I is integral (a true ideal of its order).

`IsPrime(I): RngFunOrdIdl -> BoolElt`

Returns `true` if and only if the ideal I is prime.

`IsPrincipal(I): RngFunOrdIdl -> BoolElt, FldFunElt`

Returns `true` and a generator if the fractional ideal I is principal, `false` otherwise. The function field has to be global.

Predicates on Prime Ideals

`IsInert(P): RngFunOrdIdl -> BoolElt`

Return `true` if the inertia degree of the prime ideal P is the degree of its order.

`IsInert(P, O): RngFunOrdIdl, RngFunOrd -> BoolElt`

`IsInert(P, O): RngElt, RngFunOrd -> BoolElt`

Return `true` if there is an inert ideal in the order O above the prime ideal P.

`IsRamified(P): RngFunOrdIdl -> BoolElt`

Return `true` if the ramification index of the prime ideal P is not 1.

`IsRamified(P, O): RngFunOrdIdl, RngFunOrd -> BoolElt`

`IsRamified(P, O): RngElt, RngFunOrd -> BoolElt`

Return `true` if there is a ramified ideal in the order O above the prime ideal P.

`IsSplit(P): RngFunOrdIdl -> BoolElt`

Return `true` if the prime ideal P is not the only ideal lying above the prime ideal it lies above.

`IsSplit(P, O): RngFunOrdIdl, RngFunOrd -> BoolElt`

`IsSplit(P, O): RngElt, RngFunOrd -> BoolElt`

Return `true` if there are at least 2 distinct ideals which lie in the order O above the prime ideal P.

`IsTamelyRamified(P): RngFunOrdIdl -> BoolElt`

Return whether the prime ideal P is not wildly ramified.

`IsTamelyRamified(P, O): RngFunOrdIdl, RngFunOrd -> BoolElt`

`IsTamelyRamified(P, O): RngElt, RngFunOrd -> BoolElt`

Return whether the prime ideal P is not wildly ramified in the order O.

`IsTotallyRamified(P): RngFunOrdIdl -> BoolElt`

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): RngFunOrdIdl, RngFunOrd -> BoolElt`

`IsTotallyRamified(P, O): RngElt, RngFunOrd -> BoolElt`

Return whether there are any totally ramified ideals in the order O lying above the prime ideal P.

`IsTotallySplit(P): RngFunOrdIdl -> BoolElt`

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): RngFunOrdIdl, RngFunOrd -> BoolElt`

`IsTotallySplit(P, O): RngElt, RngFunOrd -> BoolElt`

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): RngFunOrdIdl -> BoolElt`

Return whether the ramification index of the prime ideal P is 1.

`IsUnramified(P, O): RngFunOrdIdl, RngFunOrd -> BoolElt`

`IsUnramified(P, O): RngElt, RngFunOrd -> BoolElt`

Return whether all the ideals of the order O which lie above the prime ideal P are unramified.

`IsWildlyRamified(P): RngFunOrdIdl -> BoolElt`

Return whether the ramification index of the prime ideal P is a multiple of the characteristic of the residue field of P.

`IsWildlyRamified(P, O): RngFunOrdIdl, RngFunOrd -> BoolElt`

`IsWildlyRamified(P, O): RngElt, RngFunOrd -> BoolElt`

Return whether any of the ideals of the order O which lie above the prime ideal P are wildly ramified.

46.12.7 Further Ideal Operations

`I meet J: RngFunOrdIdl, RngFunOrdIdl -> RngFunOrdIdl`

The intersection of the ideals I and J.

`Gcd(I, J): RngFunOrdIdl, RngFunOrdIdl -> RngFunOrdIdl`

Given invertible ideals of an order O, returns the greatest common divisor of the ideals I and J.

`Lcm(I, J): RngFunOrdIdl, RngFunOrdIdl -> RngFunOrdIdl`

Given invertible ideals of an order O, returns the least common multiple of the ideals I and J.

`Factorization(I): RngFunOrdIdl -> [ <RngFunOrdIdl, RngIntElt> ]`

`Factorisation(I): RngFunOrdIdl -> [<RngFunOrdIdl, RngIntElt>]`

Factorization of the ideal I (as sequence of prime ideal, exponent pairs). The order must be maximal.

`Decomposition(O, p): RngFunOrd, RngElt -> [ RngFunOrdIdl ]`

`Al: MonStgElt`	Default:

`Decomposition(O, p): RngFunOrd, RngFunOrdIdl -> [ RngFunOrdIdl ]`

A sequence containing all prime ideals of the order O lying above the prime element or ideal p of any coefficient ring of O. If O is an extension of a polynomial ring over $\mathbb{Q}$ or $\mathbb{F}_q$ by a single monic integral polynomial and `Al` is set to `"Montes"` then the decomposition will be computed using the Montes algorithm [Sta18].

`Decomposition(O): RngFunOrd -> [ RngFunOrdIdl ]`

A sequence containing all prime ideals of the 'infinite' maximal order O.

`DecompositionType(O, p): RngFunOrd, RngElt -> [ <RngIntElt, RngIntElt> ]`

`DecompositionType(O, p): RngFunOrd, RngFunOrdIdl -> [ <RngIntElt, RngIntElt> ]`

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): RngFunOrd -> [ <RngIntElt, RngIntElt> ]`

Sequence of tuples of residue degrees and ramification indices of the prime ideals of the 'infinite' maximal order O.

`MultiplicatorRing(I): RngFunOrdIdl -> RngFunOrd`

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): RngFunOrd, RngFunOrdIdl -> RngFunOrd`

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 *Orders and Ideals*.

`pRadical(O, p): RngFunOrd, RngFunOrdIdl -> RngFunOrdIdl`

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): RngElt, RngFunOrdIdl -> RngIntElt`

`Valuation(I, P): RngFunOrdIdl, RngFunOrdIdl -> RngIntElt`

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. If P has been constructed using the Montes algorithm then the Montes algorithm [Sta18] will be used to compute the valuation.

`Order(I): RngFunOrdIdl -> RngFunOrd`

The order of the ideal I.

`Denominator(I): RngFunOrdIdl -> RngElt`

The "smallest" element d of the coefficient ring of the order O of the ideal I such that $dI \subseteq O$.

`Minimum(I): RngFunOrdIdl -> Any`

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: RngFunOrdIdl, Rng -> Any`

The intersection of the ideal I with a coefficient ring R of its order.

`IntegralSplit(I): RngFunOrdIdl -> RngFunOrdIdl, RngElt`

The integral ideal dI and d, where d is the denominator of the ideal I.

`Norm(I): RngFunOrdIdl -> Any`

The norm of the ideal I, as element of the coefficient field of the algebraic function field to which I belongs.

`TwoElement(I): RngFunOrdIdl -> RngElt, RngElt`

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): RngFunOrdIdl -> [ RngFunOrdElt ]`

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): RngFunOrdIdl -> [FldFunElt]`

`Basis(I, R): RngFunOrdIdl, Rng -> [ RngElt ]`

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): RngFunOrdIdl -> AlgMatElt`

Let $(b_1, \ldots, b_n)$ be the basis of the ideal I and let $(\omega_1, \ldots, \omega_n)$ be the basis of the order O. A matrix B with coefficients in the rational function field is returned such that $(b_1, \ldots, b_n) = (\omega_1, \ldots, \omega_n)B^t$.

`TransformationMatrix(I): RngFunOrdIdl -> AlgMatElt, RngElt`

Let $(b_1, \ldots, b_n)$ be the basis of the ideal I and let $(\omega_1, \ldots, \omega_n)$ 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 $(b_1, \ldots, b_n) = (\omega_1, \ldots, \omega_n)T^t/d$.

`CoefficientIdeals(I): RngFunOrdIdl -> [RngFunOrdIdl]`

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): RngFunOrdIdl -> RngFunOrdIdl`

The different of the (possibly fractional) ideal I of an order of an algebraic function field.

`Codifferent(I): RngFunOrdIdl -> RngFunOrdIdl`

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): RngFunOrdIdl -> DivFunElt`

The divisor corresponding to the ideal factorization of the ideal I.

`Divisor(I, J): RngFunOrdIdl, RngFunOrdIdl -> DivFunElt`

The divisor corresponding to the ideal factorization of the ideals I and J belonging to the 'finite' and 'infinite' maximal order.

`CoefficientHeight(I): RngFunOrdIdl -> RngIntElt`

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): RngFunOrdIdl -> RngIntElt`

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.

Example: Ideal Operations in a Relative Field (ex-bdacf7)

```
> 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:
```

(continues on next page)

(continued from previous page)

```
(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
```

Functions on Prime Ideals

RamificationIndex(I): RngFunOrdIdl -> RngIntElt

RamificationDegree(I): RngFunOrdIdl -> RngIntElt

The ramification index of the prime ideal I over the corresponding prime of its coefficient ring.

Degree(I): RngFunOrdIdl -> RngIntElt

InertiaDegree(I): RngFunOrdIdl -> RngIntElt

ResidueClassDegree(I): RngFunOrdIdl -> RngIntElt

The residue class degree (inertia degree) of the prime ideal I over the corresponding prime of its coefficient ring.

ResidueClassField(I): RngFunOrdIdl -> Rng, Map

The residue class field of the prime ideal I and the residue class mapping.

Place(I): RngFunOrdIdl -> PlcFunElt

The place corresponding to the prime ideal I, where I is defined over the 'finite' or 'infinite' maximal order.

SafeUniformizer(P): RngFunOrdIdl -> RngFunOrdElt

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.

WeakApproximation(I, V): [RngFunOrdIdl], [RngIntElt] -> FldFunElt

Compute an element in the function field of the order of the ideals in I which has valuation $V[i]$ at the prime ideal $I[i]$.

Example: Factoring an Ideal of a Maximal Order (ex-a0157f)

```
> 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)
```

46.12.8 Quotient Rings

Quotients of orders defined over maximal orders and their integral ideals can be formed resulting in an object with type `RngFunOrdRes`. Elements of such orders can be created and elementary arithmetic and predicates may be applied to them.

Operations on Quotient Rings

The creation of quotient rings and the functions which may be applied to them are described.

`quo< O | I >: RngFunOrd, RngFunOrdIdl -> RngFunOrdRes`

`quo< O | M >: RngFunOrd, ModDed -> RngFunOrdRes`

`quo< O | M >: RngFunOrd, AlgMatElt -> RngFunOrdRes`

`quo< O | S >: RngFunOrd, RngElt, ..., RngElt -> RngFunOrdRes`

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.

`quo< O | p >: RngFunOrd, RngUPolElt -> RngFunOrdRes`

`quo< O | p, r >: RngFunOrd, RngUPolElt, RngIntElt -> RngFunOrdRes`

Construct the quotient ring $Q = O/p^r$ where p is a prime polynomial in a coefficient ring of O.

`Modulus(OQ): RngFunOrdRes -> RngFunOrdIdl`

Return the denominator of the quotient ring OQ, i.e. I where $OQ = O/I$.

Example: Quotient Rings of Ideals (ex-d5cfa1)

Creation of quotient rings is shown. The orders are the same as for the ideal examples.

```
> I;
Ideal of O
Generator:
a^2
> Basis(I);
[
    x^8 + 2*x^5 + 2*x^4 + x^2 + 2*x + 1,
    a + 2*x^7 + x^4 + 2*x^3 + 2*x + 2,
    a^2,
    a^3
]
> Q := quo<Order(I) | I>;
> Q;
Quotient Ring of Ideal of O
Generators:
a^2
a^2
> Modulus(Q);
Ideal of O
Generators:
a^2
a^2
```

Elements of Quotients

Functions for elements of the quotient rings are predominantly arithmetic.

`OQ ! a: RngFunOrdRes, Elt -> RngFunOrdResElt`

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: RngFunOrdElt, RngFunOrdIdl -> RngFunOrdElt`

A canonical representative of the element a (belonging to an order O) in the quotient ring O/I.

`a * b: RngFunOrdResElt, RngFunOrdResElt -> RngFunOrdResElt`

`a + b: RngFunOrdResElt, RngFunOrdResElt -> RngFunOrdResElt`

`a - b: RngFunOrdResElt, RngFunOrdResElt -> RngFunOrdResElt`

`a / b: RngFunOrdResElt, RngFunOrdResElt -> RngFunOrdResElt`

`- a: RngFunOrdResElt -> RngFunOrdResElt`

`a ^ n: RngFunOrdResElt, RngIntElt -> RngFunOrdResElt`

`a eq b: RngFunOrdResElt, RngFunOrdResElt -> BoolElt`

`a ne b: RngFunOrdResElt, RngFunOrdResElt -> BoolElt`

`IsZero(a): RngFunOrdResElt -> BoolElt`

Returns `true` if and only if the quotient ring element a is the zero element of the quotient ring OQ.

`IsOne(a): RngFunOrdResElt -> BoolElt`

Returns `true` if and only if the quotient ring element a is the one element of the quotient ring OQ.

`IsMinusOne(a): RngFunOrdResElt -> BoolElt`

Returns `true` if and only if the quotient ring element a is the minus one element of the quotient ring OQ.

`IsUnit(a): RngFunOrdResElt -> BoolElt`

Returns `true` if and only if the quotient ring element a has an inverse in the quotient ring OQ.

`Eltseq(a): RngFunOrdResElt -> []`

`ElementToSequence(a): RngFunOrdResElt -> []`

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.

46.13 Places

46.13.1 Creation of Structures

`Places(F): FldFun -> PlcFun`

The set of places of the algebraic function field F/k.

46.13.2 Creation of Elements

General Function Field Places

`Decomposition(F, P): FldFunG, PlcFunElt -> [ PlcFunElt ]`

`Al: MonStgElt`	Default:

A sequence containing all places of F/k lying above the place P of any coefficient field of F. The function field F must be a finite extension of $k(x)$. If F is an extension of a rational function field over $\mathbb{Q}$ or $\mathbb{F}_q$ by a single monic integral polynomial, P is a finite place and `Al` is set to `"Montes"` then the Montes algorithm [Sta18] will be used to compute the decomposition.

`DecompositionType(F, P): FldFun, PlcFunElt -> [ <RngIntElt, RngIntElt> ]`

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): FldFunElt -> [ PlcFunElt ]`

`Zeros(a): RngFunOrdElt -> [ PlcFunElt ]`

A sequence containing all zeros of the algebraic function a.

`Poles(a): FldFunElt -> [ PlcFunElt ]`

`Poles(a): RngFunOrdElt -> [ PlcFunElt ]`

A sequence containing all poles of the algebraic function a.

`S ! I: PlcFun, RngFunOrdIdl -> PlcFunElt`

`Place(I): RngFunOrdIdl -> PlcFunElt`

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): DivFunElt -> [ PlcFunElt ], [ RngIntElt ]`

`Support(P): PlcFunElt -> [ PlcFunElt ], [ RngIntElt ]`

Sequences containing the places and exponents occurring in the divisor D.

`AssignNames(~P, s): PlcFunElt, [ MonStgElt ]`

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): FldFun -> [PlcFunElt]`

The infinite places of the function field F.

Global Function Field Places

In this section F/k denotes a global function field.

`HasPlace(F, m): FldFun, RngIntElt -> PlcFunElt`

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): FldFun, RngIntElt -> BoolElt, PlcFunElt`

Returns `true` and a random place of degree m in the function field F/k or (`false` if there are none).

`RandomPlace(F, m): FldFun, RngIntElt -> PlcFunElt`

Returns a random place of degree m in the function field F/k or throws an error if there is none.

`Places(F, m): FldFun, RngIntElt -> SeqEnum[PlcFunElt]`

A sequence containing the places of degree m of the function field F/k.

Example: Creating Places (ex-92c345)

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)
```

46.13.3 Related Structures

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): PlcFun -> FldFun

The corresponding function field of the set of places S.

DivisorGroup(F): FldFunG -> DivFun

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.

46.13.4 Structure Invariants

General Function Fields

WeierstrassPlaces(F): FldFunG -> [PlcFunElt]

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*.

Global Function Fields

In this section F/k denotes a global function field.

NumberOfPlacesOfDegreeOneOverExactConstantField(F, m): FldFun, RngIntElt -> RngIntElt

NumberOfPlacesOfDegreeOneECF(F, m): FldFunG, RngIntElt -> RngIntElt

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): FldFun, RngIntElt -> RngIntElt

NumberOfPlacesOfDegreeOneECFBound(F, m): FldFunG, RngIntElt -> RngIntElt

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): FldFunG, RngIntElt -> RngIntElt

NumberOfPlacesDegECF(F, m): FldFunG, RngIntElt -> RngIntElt

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.

46.13.5 Structure Predicates

S1 eq S2: PlcFun, PlcFun -> BoolElt

S1 ne S2: PlcFun, PlcFun -> BoolElt

46.13.6 Element Operations

Parent and Category

Parent(P): PlcFunElt -> PlcFun

Category(P): PlcFunElt -> Cat

Arithmetic Operators

`- P: PlcFunElt -> DivFunElt`

`P1 + P2: PlcFunElt, PlcFunElt -> DivFunElt`

`P1 - P2: PlcFunElt, PlcFunElt -> DivFunElt`

`k * P: RngIntElt, PlcFunElt -> DivFunElt`

`P div k: PlcFunElt, RngIntElt -> DivFunElt`

`P mod k: PlcFunElt, RngIntElt -> DivFunElt`

`Quotrem(P, k): PlcFunElt, RngIntElt -> DivFunElt, DivFunElt`

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.

Equality and Membership

`P1 eq P2: PlcFunElt, PlcFunElt -> BoolElt`

`P1 ne P2: PlcFunElt, PlcFunElt -> BoolElt`

`P in S: PlcFunElt, PlcFun -> BoolElt`

`P notin S: PlcFunElt, PlcFun -> BoolElt`

Predicates on Elements

`IsFinite(P): PlcFunElt -> BoolElt`

Returns `true` if the place P is a 'finite' place.

`IsWeierstrassPlace(P): PlcFunElt -> BoolElt`

`IsWeierstrassPlace(F, P): FldFunG, PlcFunElt -> BoolElt`

Whether the degree one place P is a Weierstraß place of its function field F. See the description of *WeierstrassPlaces*.

Other Element Operations

`FunctionField(P): PlcFunElt -> FldFun`

The function field that corresponds to the place P.

`Degree(P): PlcFunElt -> RngIntElt`

The degree of the place P over the constant field of definition k.

`RamificationIndex(P): PlcFunElt -> RngIntElt`

`RamificationDegree(P): PlcFunElt -> RngIntElt`

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): PlcFunElt -> RngIntElt`

`ResidueClassDegree(P): PlcFunElt -> RngIntElt`

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): PlcFunElt -> RngElt`

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): PlcFunElt -> Rng, Map`

The residue class field of the place P and the map from the order of the place into the field.

`Evaluate(a, P): RngElt, PlcFunElt -> RngElt`

Evaluate the algebraic function a at the place P. If it is not defined at P, infinity is returned.

`Lift(a, P): RngElt, PlcFunElt -> FldFunElt`

`Lift(i, P): Infty, PlcFunElt -> FldFunElt`

Lift the element a of the residue class field of the place P (including infinity) to an algebraic function.

`TwoGenerators(P): PlcFunElt -> FldFunGElt, FldFunGElt`

Two algebraic functions having the place P as their unique common zero.

`LocalUniformizer(P): PlcFunElt -> FldFunGElt`

`UniformizingElement(P): PlcFunElt -> FldFunGElt`

A local uniformizing parameter at the place P.

`Valuation(a, P): FldFunElt, PlcFunElt -> RngIntElt`

The valuation of the element a at the place P.

`Ideal(P): PlcFunElt -> RngFunOrdIdl`

Create a prime ideal corresponding to the place P.

`Norm(P): PlcFunElt -> DivFunElt`

The divisor of the norm of the ideal of the place P.

Example: Counting Places of a Given Degree (ex-bbb5e4)

```
> 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
```

46.13.7 Completion at Places

`Completion(F, p): FldFun, PlcFunElt -> RngSerLaur, Map`

`Completion(O, p): RngFunOrd, PlcFunElt -> RngSerPow, Map`

```
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.

46.14 Divisors

Divisors are represented in at least one of 4 ways. They can be represented

- as a linear combination of places (as given by *Support*),
- by an ideal of a finite maximal order and an ideal of a corresponding infinite maximal order (as given by *Ideals*)
- in a reduced representation (as given by *Reduction*) or
- as a complementary divisor to another divisor (as given by *ComplementaryDivisor*).

While the representation as a linear combination of places can be efficient for divisor arithmetic, the reduced representation is more efficient for computations like *Dimension*. Computing the *Support* of a divisor will ensure it has a representation as a linear combination stored. When divisors are combined in binary operations extra representations may need to be computed.

46.14.1 Creation of Structures

`DivisorGroup(F): FldFun -> DivFun`

Create the group of divisors of the algebraic function field F/k.

46.14.2 Creation of Elements

`Divisor(P): PlcFunElt -> DivFunElt`

`Div ! P: DivFun, PlcFunElt -> DivFunElt`

`1 * P: PlcFunElt -> DivFunElt`

Given a place P in a function field, return the prime divisor $1 * P$.

`Div ! a: DivFun, RngElt -> DivFunElt`

`Divisor(a): FldFunElt -> DivFunElt`

`Divisor(a): RngFunOrdElt -> DivFunElt`

Given an algebraic function a, return the principal divisor (a).

`Div ! I: DivFun, RngFunOrdIdl -> DivFunElt`

`Divisor(I): RngFunOrdIdl -> DivFunElt`

The divisor corresponding to the factorization of the ideal I.

`Divisor(I, J): RngFunOrdIdl, RngFunOrdIdl -> DivFunElt`

The divisor corresponding to the ideal factorization of the ideals I and J belonging to the 'finite' and 'infinite' maximal order.

`Identity(G): DivFun -> DivFunElt`

`Id(G): DivFun -> DivFunElt`

Given the group G of divisors of a function field, return the zero divisor.

`CanonicalDivisor(F): FldFunG -> DivFunElt`

A canonical divisor of the function field F/k.

`DifferentDivisor(F): FldFunG -> DivFunElt`

The different divisor of the underlying extension of the function field $F/k(x)$.

`AssignNames(~D, s): DivFunElt, [ MonStgElt ]`

Change the print name employed when displaying D to be the contents of s which must have length 1 in this case.

46.14.3 Related Structures

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): DivFun -> FldFun`

Given the group G of divisors of a function field F/k, return F.

`Places(F): FldFun -> PlcFun`

The set of places of the algebraic function field F/k.

46.14.4 Structure Invariants

`NumberOfSmoothDivisors(n, m, P): RngIntElt, RngIntElt, SeqEnum[RngElt] -> RngElt`

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 [Hes99].

`DivisorOfDegreeOne(F): FldFunG -> DivFunElt`

A divisor of degree one over the exact constant field of the global function field F/k.

46.14.5 Structure Predicates

`Div1 eq Div2: DivFun, DivFun -> BoolElt`

`Div1 ne Div2: DivFun, DivFun -> BoolElt`

46.14.6 Element Operations

Arithmetic Operators

`- D: DivFunElt -> DivFunElt`

`D1 + D2: DivFunElt, DivFunElt -> DivFunElt`

`D1 - D2: DivFunElt, DivFunElt -> DivFunElt`

`k * D: RngIntElt, DivFunElt -> DivFunElt`

`D div k: DivFunElt, RngIntElt -> DivFunElt`

`D mod k: DivFunElt, RngIntElt -> DivFunElt`

`P + D: PlcFunElt, DivFunElt -> DivFunElt`

`D + P: DivFunElt, PlcFunElt -> DivFunElt`

`D - P: DivFunElt, PlcFunElt -> DivFunElt`

`P - D: PlcFunElt, DivFunElt -> DivFunElt`

`Quotrem(D, k): DivFunElt, RngIntElt -> DivFunElt, DivFunElt`

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): DivFunElt, DivFunElt -> DivFunElt`

`Gcd(D1, D2): DivFunElt, DivFunElt -> DivFunElt`

`GreatestCommonDivisor(D1, D2): DivFunElt, DivFunElt -> DivFunElt`

The greatest common divisor of the divisors $D1$ and $D2$.

`LCM(D1, D2): DivFunElt, DivFunElt -> DivFunElt`

`Lcm(D1, D2): DivFunElt, DivFunElt -> DivFunElt`

`LeastCommonMultiple(D1, D2): DivFunElt, DivFunElt -> DivFunElt`

The least common multiple of the divisors $D1$ and $D2$.

Equality, Comparison, and Membership

D1 **eq** D2: DivFunElt, DivFunElt -> BoolElt

D1 **ne** D2: DivFunElt, DivFunElt -> BoolElt

D1 **le** D2: DivFunElt, DivFunElt -> BoolElt

D1 **lt** D2: DivFunElt, DivFunElt -> BoolElt

D1 **ge** D2: DivFunElt, DivFunElt -> BoolElt

D1 **gt** D2: DivFunElt, DivFunElt -> BoolElt

D **in** Div: DivFunElt, DivFun -> BoolElt

D **notin** Div: DivFunElt, DivFun -> BoolElt

Predicates on Elements

IsZero(D): DivFunElt -> BoolElt
IsEffective(D): DivFunElt -> BoolElt
IsPositive(D): DivFunElt -> BoolElt
IsSpecial(D): DivFunElt -> BoolElt
IsPrincipal(D): DivFunElt -> BoolElt, FldFunElt
IsCanonical(D): DivFunElt -> BoolElt, DiffFunElt

Returns **true** iff the divisor D is canonical and a differential having D as its divisor.

Example: Divisors on a Relative Function Field (ex-749efd)

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)
```

(continues on next page)

(continued from previous page)

```
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,
```

(continues on next page)

(continued from previous page)

```
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
```

Other Element Operations

FunctionField(D): DivFunElt -> FldFun

Given a divisor D, return the function field.

Degree(D): DivFunElt -> RngIntElt

The degree of the divisor D over k, the constant field of definition.

Support(D): DivFunElt -> [PlcFunElt]

A sequence containing the places occurring in the divisor D.

Numerator(D): DivFunElt -> DivFunElt

ZeroDivisor(D): DivFunElt -> DivFunElt

The numerator of the divisor D.

Denominator(D): DivFunElt -> DivFunElt

`PoleDivisor(D): DivFunElt -> DivFunElt`

The denominator of the divisor D.

`Ideals(D): DivFunElt -> RngFunOrdIdl, RngFunOrdIdl`

Create two ideals of the 'finite' and 'infinite' maximal order respectively corresponding to the divisor D.

`Norm(D): DivFunElt -> DivFunElt`

The divisor of the norms of the ideals of the divisor D.

`FiniteSplit(D): DivFunElt -> DivFunElt, DivFunElt`

`FiniteDivisor(D): DivFunElt -> DivFunElt`

`InfiniteDivisor(D): DivFunElt -> DivFunElt`

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): DivFunElt -> RngIntElt`

The dimension of the Riemann-Roch space $\mathcal{L}(D)$ of the divisor D over k, the constant field of definition.

`IndexOfSpeciality(D): DivFunElt -> RngIntElt`

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): DivFunElt -> [RngElt], [RngIntElt]`

```
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 $d_i \in \mathbb{Z}$ 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 [Hes99].

`Basis(D : parameters): DivFunElt -> [ FldFunElt ]`

```
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): DivFunElt -> ModFld, Map`

`RiemannRochSpace(P): PlcFunElt -> ModFld, Map`

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): DivFunElt, PlcFunElt -> RngIntElt`

The exponent of the place P in the divisor D.

`Reduction(D): DivFunElt -> DivFunElt, RngIntElt, DivFunElt, FldFunElt`

`Reduction(D, A): DivFunElt, DivFunElt -> DivFunElt, RngIntElt, DivFunElt, FldFunElt`

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): DivFunElt, PlcFunElt -> SeqEnum[RngIntElt]`

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 $\dim\big(D + (m-1)P\big) = \dim(D + mP)$ 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): DivFunElt -> SeqEnum[RngIntElt]`

`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*.

Example: Riemann-Roch Space of a Divisor (ex-bddf6b)

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);
```

(continues on next page)

(continued from previous page)

```
> 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: Algebraic Relation for Sine and Cosine (ex-3b8680)

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: Sine and Cosine via Exponentials (ex-f4fe11)

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): DivFunElt -> DivFunElt`

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 Wronskian 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): DivFunElt -> [PlcFunElt]`

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): DivFunElt, PlcFunElt -> BoolElt`

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): DivFunElt -> [RngIntElt]`

`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 $\nu_1 \leq \cdots \leq \nu_l \in \mathbb{Z}^{\geq 0}$ 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): DivFunElt -> DivFunElt`

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): DivFunElt -> [DiffFunElt]`

A basis of the space of differentials of the divisor D. See *DifferentialBasis* for details.

`DifferentialSpace(D): DivFunElt -> ModFld, Map`

A vector space and the isomorphism from this space to the differential space of the divisor D.

`Parametrization(F, D): FldFun, DivFunElt -> FldFunElt, [FldFunRatUElt]`

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.

46.14.7 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 $\mathbb{Z}$ 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 $\mathbb{Z}/c_1\mathbb{Z} \times \cdots \times \mathbb{Z}/c_{2g}\mathbb{Z} \times \mathbb{Z}$ with integers $c_1 | \dots | c_{2g}$ 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 [Hes99].

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): RngIntElt, RngIntElt -> RngIntElt`

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): FldFunG -> RngIntElt`

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): FldFunG, FldReElt -> FldReElt`

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

$$\left|\log\left(h\,/\,q^{g}\right)-\sum_{r=1}^{b}q^{-r}/r\left(N_r-(q^r+1)\right)\right|\;\leq\;2gq^{-b/2}/\left((q^{1/2}-1)(b+1)\right)$$

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): RngIntElt, RngIntElt, FldReElt -> RngIntElt`

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): FldFun -> GrpAb, Map, Map`

```
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): FldFun -> SeqEnum`

```
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): FldFun -> RngIntElt`

The order of the group of divisor classes of degree zero of the function field F/k.

Example: Divisor Class Group Computations (ex-caa2d2)

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): FldFun -> GrpAb, Map

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): FldFunElt -> BoolElt

Whether the function field element a is a global unit, i.e. a constant (equivalent to `IsConstant`).

IsGlobalUnitWithPreimage(a): FldFunElt -> BoolElt, GrpAbElt

Returns `true` and the preimage of the function field element a in the global unit group, `false` otherwise.

PrincipalDivisorMap(F): FldFunG -> Map

The map from the multiplicative group of the function field to the group of divisors.

ClassGroupExactSequence(F): FldFunG -> Map, Map, Map

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): SetEnum[PlcFunElt] -> GrpAb, Map

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): FldFunElt, SetEnum[PlcFunElt] -> BoolElt`

Returns `true` if the function field element a is an S-unit for the sequence of places S, `false` otherwise.

`IsSUnitWithPreimage(a, S): FldFunElt, SetEnum[PlcFunElt] -> BoolElt, GrpAbElt`

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): SetEnum[PlcFunElt] -> RngIntElt`

The S-Regulator for the sequence of places S.

`SPrincipalDivisorMap(S): SetEnum[PlcFunElt] -> Map`

The map from the multiplicative group of the function field to the group of divisors (mod places in the sequence S).

`IsSPrincipal(D, S): DivFunElt, SetEnum[PlcFunElt] -> BoolElt, FldFunElt`

Returns `true` and a generator if the divisor D is principal modulo places in the sequence S, `false` otherwise

`SClassGroup(S): SetEnum[PlcFunElt] -> GrpAb, Map, Map`

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): SetEnum[PlcFunElt] -> Map, Map, Map`

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): SetEnum[PlcFunElt] -> SeqEnum`

Computes a sequence of integers containing the Abelian invariants of the S-class group for the sequence of places S.

`SClassNumber(S): SetEnum[PlcFunElt] -> RngIntElt`

The order of the torsion part of the S-class group for the sequence of places S.

`ClassGroupPRank(F): FldFunG -> RngIntElt`

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): FldFunG -> RngIntElt`

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): DivFunElt, DivFunElt, RngIntElt -> RngElt`

The Tate–Lichtenbaum pairing $Cl_0[m] \times Cl_0/mCl_0 \to k$ for coprime divisors $D1$ and $D2$.

Example: Tate-Lichtenbaum Pairing of Divisors (ex-7f9ab0)

```
> 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;
```

(continues on next page)

(continued from previous page)

```
> 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
```

46.15 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.

46.15.1 Creation of Structures

`DifferentialSpace(F): FldFunG -> DiffFun`

The space of differentials of the algebraic function field F/k.

46.15.2 Creation of Elements

The simplest ways of creating a differential are given below.

`Differential(a): FldFunGElt -> DiffFunElt`

`Differential(a): RngFunOrdElt -> DiffFunElt`

Create the differential $d(a)$ of the function field or order element a.

`Identity(D): DiffFun -> DiffFunElt`

`Id(D): DiffFun -> DiffFunElt`

The identity differential of the space of differentials D.

`IsCanonical(D): DivFunElt -> BoolElt, DiffFunElt`

Returns `true` iff the divisor D is canonical and a differential having D as its divisor.

46.15.3 Related Structures

`FunctionField(D): DiffFun -> FldFun`

The function field of the differentials in the space D.

`FunctionField(d): DiffFunElt -> FldFun`

The function field of the differential d.

46.15.4 Subspaces

`SpaceOfDifferentialsFirstKind(F): FldFunG -> ModFld, Map`

`SpaceOfHolomorphicDifferentials(F): FldFunG -> ModFld, Map`

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): FldFunG -> SeqEnum[DiffFunElt]`

`BasisOfHolomorphicDifferentials(F): FldFunG -> SeqEnum[DiffFunElt]`

A basis of the space of differentials of the first kind (holomorphic differentials) of the function field F/k.

`DifferentialBasis(D): DivFunElt -> [DiffFunElt]`

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): DivFunElt -> ModFld, Map`

A vector space and the isomorphism from this space to the differential space of the divisor D.

Example: Differential Space of a Divisor (ex-b69721)

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) ]
```

46.15.5 Structure Predicates

`D1 eq D2: DiffFun, DiffFun -> BoolElt`

Return `true` if the spaces of differentials $D1$ and $D2$ are the same.

46.15.6 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.

Arithmetic Operators

`r * x: RngElt, DiffFunElt -> DiffFunElt`

`x * r: DiffFunElt, RngElt -> DiffFunElt`

`x + y: DiffFunElt, DiffFunElt -> DiffFunElt`

`- x: DiffFunElt -> DiffFunElt`

`x - y: DiffFunElt, DiffFunElt -> DiffFunElt`

`x / y: DiffFunElt, DiffFunElt -> FldFunGElt`

`x / r: DiffFunElt, RngElt -> DiffFunElt`

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 $x = ry$.

Equality and Membership

`x eq y: DiffFunElt, DiffFunElt -> BoolElt`

Returns `true` if x and y are the same differential.

`x in D: Any, DiffFun -> BoolElt`

Returns `true` if x is in the space of differentials D.

Predicates on Elements

`IsExact(d): DiffFunElt -> BoolElt, FldFunGElt`

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): DiffFunElt -> BoolElt`

Return `true` if d is the zero differential.

Functions on Elements

`Valuation(d, P): DiffFunElt, PlcFunElt -> RngIntElt`

The valuation of the differential d at the place P.

`Divisor(d): DiffFunElt -> DivFunElt`

The divisor (d) of the differential d.

`Residue(d, P): DiffFunElt, PlcFunElt -> RngElt`

The residue of the differential d at the place P, where P must (currently) have degree one.

Example: Functions on Differentials (ex-7c77de)

```
> 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): SeqEnum[ DiffFunElt ], Rng -> Mod, Map, SeqEnum[ ModElt ]`

```
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): SeqEnum[ DiffFunElt ], Rng -> ModTupRng`

`Relations(L, R, m): SeqEnum[ DiffFunElt ], Rng, RngIntElt -> ModTupRng`

The module of R-linear relations between the differentials of the sequence L. The integer 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 $M := \{ \sum_{i=1}^{m} v_i a_i \,|\, v = (v_i)_i \in V \}$. The function tries to compute a generating system of V such that the corresponding generating system of M consists of "small" elements.

Example: Differentials as a Module (ex-3e031a)

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): DiffFunElt -> DiffFunElt`

`Cartier(b, r): DiffFunElt, RngIntElt -> DiffFunElt`

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.

Other

`CartierRepresentation(F): FldFunG -> AlgMatElt, SeqEnum[DiffFunElt]`

`CartierRepresentation(F, r): FldFunG, RngIntElt -> AlgMatElt, SeqEnum[DiffFunElt]`

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 $r \in \mathbb{Z}^{\geq 1}$. 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: Trivial Cartier Action on Differentials (ex-78c02f)

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
[]
```

46.16 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 [PGS00] or the chapter by F. Heß in [Bla05]). There is a corresponding function `WeilDescent(E, k, c): FldFun, FldFin, FldFinElt → CrvPln, Map` 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): FldFun, FldFin -> FldFunG, Map`

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) $Trace_{K/\mathbb{F}_p}(a) = 0$

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(E, k, c): FldFun, FldFin, FldFinElt -> CrvPln, Map` 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): FldFin, FldFin, FldFin -> FldFun`

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): FldFun, FldFin -> RngIntElt`

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): FldFun, FldFin -> RngIntElt`

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): RngElt, RngUPolElt, Map -> RngElt`

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 $\mathbb{F}_p$-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: GHS Weil Descent `(ex-469a62)`

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);
```

(continues on next page)

(continued from previous page)

```
> 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)
```

46.17 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:

$\mathbb{F}_2$:
degrees 2, 3

$\mathbb{F}_3$:
degree 2

$\mathbb{F}_4$:
degrees 2, 3, 5

$\mathbb{F}_5$:
degrees 2, 3, 4, 8

$\mathbb{F}_7$:
degree 9

$\mathbb{F}_{11}$:
degree 3

$\mathbb{F}_{13}$:
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< D | : parameters>* constructor: The genus; the number of degree one places; the class number; and the class group.

46.17.1 Creation

FunctionFieldDatabase(q, d): RngIntElt, RngIntElt -> DB

Returns a database object for the function fields of degree d over $\mathbb{F}_q$.

sub< D | : parameters>: DB -> DB

```
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.

46.17.2 Access

BaseField(D): DB -> FldFin

CoefficientField(D): DB -> FldFin

Returns the finite field underlying each function field in the database.

Degree(D): DB -> RngIntElt

Returns the degree of each function field in the database.

D: DB -> RngIntElt

NumberOfFields(D): DB -> RngIntElt

Returns the number of function fields stored in the database.

FunctionFields(D): DB -> [FldFunG]

Returns the sequence of function fields stored in the database.

Example: Genus Distribution in the Function Field Database (ex-f1c514)

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 ]
```

46.18 The Montes Algorithm

The Montes algorithm can be considered as a factorization algorithm for polynomials over local fields. For a defining polynomial f of a function field K and a prime polynomial p the Montes algorithm produces OM representations for all prime ideals of K lying of over p, which parametrize the irreducible factors of f over the p-adic series. The OM representation for prime ideals leads to an OM presentation of fractional ideals. The basic algorithms are explained in [Bau14, Bau16a, GMN13] These OM representations allow the construction of ideals and divisors without the computation of a maximal order, which the

construction of ideals of type `RngFunOrdIdl` require. The Montes algorithm can also be used to efficiently compute integral bases which can be use to construct maximal orders.

This section explains the use of the Montes Algorithm. For how to use the Montes algorithm to compute maximal orders of function fields see Section *Construction of Orders of Algebraic Function Fields*, and decomposition of primes in function fields see Sections *Further Ideal Operations* and *Creation of Elements*.

`Montes(f, p): RngUPolElt, RngUPolElt -> SeqEnum, SeqEnum, RngIntElt`

Field: BoolElt	Default: false

Let f be a monic polynomial with polynomial coefficients and p be a prime polynomial. The Montes algorithm outputs the triple:

1. A sequence of OM representations for the p-adic irreducible factors of f.
2. A sequence of intervals with the position of the prime ideals in each disconnected tree of OM representations.
3. The p-index of f.

If the parameter `Field` is set to `true` the ψ polynomial at the last level of each type of each OM representation will be computed.

Example: Montes Algorithm for a Polynomial (ex-531fa2)

Here is an example of how to use it:

```
> k := GF(13);
> A<t> := PolynomialRing(k);
> Ax<x> := PolynomialRing(A);
> f := x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 +
>      (t^3 + 3*t^2 + 3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t;
> p := t;
> Montes(f,p);
[
    ,

]
[
    [ 1 ],
    [ 2 ]
]
0
```

The Montes algorithm can also be called for a function field K if it is defined by a monic polynomial with polynomial coefficients.

`Montes(K, p): FldArith, RngElt`

The Montes algorithm is applied to the defining polynomial of the function field K at the prime p.

1. A sequence of OM representations for the prime ideals of K is stored in `K`PrimeIdeals[p]`.
2. The p-adic valuation of the index $[\mathbb{Z}_K : \mathbb{Z}[\theta]]$ is stored in `K`LocalIndex[p]`

Example: Montes Algorithm for a Function Field (ex-bec397)

Here's an example of how to use it for a field :

```
> k := GF(13);
> A<t> := PolynomialRing(k);
> Ax<x> := PolynomialRing(A);
```

(continues on next page)

(continued from previous page)

```
> f := x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 +
>     (t^3 + 3*t^2 + 3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t;
> L := FunctionField(f);
> p := t+1;
> Montes(L,p);
> L`PrimeIdeals[p];
[
    OM prime ideal over t + 1
    of Algebraic function field defined over Univariate rational function field
    over GF(13) by
    x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3 +
    3*t^2 + 3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t
    having residual degree   1
    and ramification index   1
    Last phi polynomial is x + 12,
    OM prime ideal over t + 1
    of Algebraic function field defined over Univariate rational function field
    over GF(13) by
    x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3 +
    3*t^2 + 3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t
    having residual degree   4
    and ramification index   1
    Last phi polynomial is x^4 + x^3 + x^2 + x + 1
]
> L`LocalIndex[p];
0
```

SFL(P, s): OMIdl, RngIntElt

Perform the single factor lifting algorithm, [GNP12], on the prime ideal P in OM representation until the slope is at least s.

Example: Refining an OM Prime Ideal Approximation (ex-8aed80)

```
> k := GF(13);
> A<t> := PolynomialRing(k);
> Ax<x> := PolynomialRing(A);
> f := x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3
> + 3*t^2 + 3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t;
> L := FunctionField(f);
> p := t+1;
> Montes(L,p);
> P := L`PrimeIdeals[p,1];
> P`Type[#P`Type]`slope;
1
> P`Type[#P`Type]`Phi; // the approximation
x + 12
> SFL(~P,10:update := true);
> P`Type[#P`Type]`slope;
10
> P`Type[#P`Type]`Phi; // the approximation
x + 6*t^9 + 2*t^8 + 11*t^7 + 3*t^6 + 2*t^5 + 2*t^4 + 11*t^3 + 3*t^2 + 12*t + 5
```

SetUseMontes(f): BoolElt

SetUseMontes(t, f): Cat, BoolElt

Set whether the Montes algorithm is to be used for maximal order and prime decomposition calculations. The type t can be `FldFun`. This setting can be overwritten by setting the `Al` parameter of the *MaximalOrder* and *Decomposition* intrinsics to `"Montes"`.

`GetUseMontes(t): Cat -> BoolElt`

Return whether the Montes algorithm will be used by default for computations of maximal orders and decompositions for types inheriting from type t, where t can be `FldFun`.

`SetVerbose("Montes", v): MonStgElt, RngIntElt`

Set the level for verbose printing for the Montes algorithm to be $v \leq 3$.

46.19 Ideals in OM Representation

Ideals of type `RngFunOrdIdl` can have an OM representation computed from them. Similarly ideals constructed from the Montes algorithm in OM representation can be converted into ideals of type `RngFunOrdIdl`.

`Ideal(I): OMIdl -> RngOrdIdl`

Translates ideal I in OM representation into a Magma representation.

`OMRepresentation(I): RngFunOrdIdl -> OMIdl`

`OMRepresentation(I): RngOrdFracIdl -> OMIdl`

Computes the OM representation of the ideal I in a function field.

`OMRepresentation(L, S): FldArith, [FldArithElt] -> OMIdl`

`OMRepresentation(L, a): FldArith, FldArithElt -> OMIdl`

`OMRepresentation(L, a): FldArith, RngElt -> OMIdl`

Given a function field L and an element a or sequence S of elements in L, construct, in OM representation, the ideal generated by these elements.

Example: Converting OM Prime Ideals to Ideals (ex-c49fbd)

```
> F<t> := FunctionField(Rationals());
> P<x> := PolynomialRing(F);
> L<a> := ext<F | x^2 + t>;
> Montes(L, Numerator(t + 1));
> Ideal(L`PrimeIdeals[t+1][1]);
Prime Ideal of Maximal Equation Order of L over Univariate Polynomial Ring in t
over Rational Field
Generators:
t + 1
a - 1
> OMRepresentation(L, [t^2 + 6*t + 5, (t + 5)*a + t + 5]);
OM ideal of the field Algebraic function field defined over Univariate rational
function field over Rational Field by
x^2 + t
generated by [
t^2 + 6*t + 5,
(t + 5)*a + t + 5
]

> Ideal($1);
Ideal of Maximal Equation Order of L over Univariate Polynomial Ring in t over
Rational Field
Generators:
t^2 + 6*t + 5
(-1/32*t^3 - 9/32*t^2 - 15/32*t + 25/32)*a + 1/32*t^3 + 5/32*t^2 + 7/32*t +
    35/32
```

46.19.1 Ideal Arithmetic

`I + J: OMIdl, OMIdl -> OMIdl`

`I * J: OMIdl, OMIdl -> OMIdl`

`I / J: OMIdl, OMIdl -> OMIdl`

The sum, product or quotient of ideals I and J in OM representation.

`I ^ n: OMIdl, RngIntElt -> OMIdl`

The n-th power of the ideal I in OM representation.

Example: Arithmetic of OM Ideals (ex-0a493e)

```
> k := GF(13);
> A<t> := PolynomialRing(k);
> Ax<x> := PolynomialRing(A);
> f := x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 +
>     (t^3 + 3*t^2 + 3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t;
> L := FunctionField(f);
> I := OMRepresentation(L,L.1+1);
> I;
OM ideal of the field Algebraic function field defined over Univariate rational
function field over GF(13) by
x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3 + 3*t^2 +
3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t
generated by [
$.1 + 1
]

> Montes(L, t + 1);
> P := L`PrimeIdeals[t+1, 1];
>      J := P^-2;
> I*J;
OM ideal of the field Algebraic function field defined over Univariate rational
function field over GF(13) by
x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3 + 3*t^2 +
3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t
having the factorization [
<t + 1, 1, -2>,
<t^4 + 10*t^3 + 4*t^2 + t + 7, 1, 1>
]

>  I^2;
OM ideal of the field Algebraic function field defined over Univariate rational
function field over GF(13) by
x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3 + 3*t^2 +
3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t
having the factorization [
<t^4 + 10*t^3 + 4*t^2 + t + 7, 1, 2>
]

> I/J;
OM ideal of the field Algebraic function field defined over Univariate rational
function field over GF(13) by
x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3 + 3*t^2 +
3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t
having the factorization [
<t + 1, 1, 2>,
<t^4 + 10*t^3 + 4*t^2 + t + 7, 1, 1>
```

(continues on next page)

(continued from previous page)

```
]

> I+J;
OM ideal of the field Algebraic function field defined over Univariate rational
function field over GF(13) by
x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3 + 3*t^2 +
3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t
having the factorization  []
generated by [
1,
0
]
```

46.19.2 Ideal Predicates

IsOne(I): OMIdl -> BoolElt

Given an ideal I in OM representation returns whether the ideal is generated by the 1 element of the field.

IsZero(I): OMIdl -> BoolElt

Given an ideal I in OM representation returns whether the ideal contains only the 0 element of the field.

I **eq** J: OMIdl, OMIdl -> BoolElt

Given two ideals I and J in OM representation returns whether these ideals are the same.

a **in** I: RngElt, OMIdl -> BoolElt

Given an element a coercible into the field containing the ideal I in OM representation return whether a is contained in the ideal.

I **subset** J: OMIdl, OMIdl -> BoolElt

Given two ideals I and J in OM representation returns whether I is contained in J.

IsPrime(I): OMIdl -> BoolElt

Given an ideal I in OM representation returns whether only the 1 ideal and I contain I.

IsIntegral(I): OMIdl -> BoolElt

Given an ideal I in OM representation returns whether all elements in the ideal I are integral.

46.20 Ideal Operations

pIntegralBasis(I, p): OMIdl, RngElt -> SeqEnum

Given an ideal I in OM representation and a prime element p in the field containing I, compute a p-integral basis for I.

SIntegralBasis(I, S): OMIdl, SeqEnum -> SeqEnum

SIntegralBasis(I, S): RngOrdFracIdl, [RngIntElt] -> SeqEnum

SIntegralBasis(I, S): RngFunOrdIdl, [RngUPolElt] -> SeqEnum

Given an ideal I in either representation and a sequence S of primes in the field containing I, compute an S-integral basis of I for the given set of primes S.

Basis(I): OMIdl -> SeqEnum

```
HNF      : BoolElt                  Default: false
Separated: BoolElt                  Default: false
```

Given an ideal I in OM representation, return a basis for I.

If `HNF` is set to `true` a triangular basis in Hermite form is returned.

If `Separated` is set to `true` the basis will be returned as a sequence of numerators and denominators.

Example: Integral Basis of an OM Ideal (ex-70e335)

```
> k := GF(13);
> A<t> := PolynomialRing(k);
> Ax<x> := PolynomialRing(A);
> f := x^4 + (4*t + 4)*x^3 + (6*t^2 + 12*t + 6)*x^2 + (4*t^3 + 12*t^2 + 12*t
>                             + 4)*x + t^4 + 5*t^3 + 6*t^2 + 4*t + 1;
> L := FunctionField(f);
> p := t;
> I := OMRepresentation(L,[L.1,p^12]);
> pIntegralBasis(I,p);
[
    1,
    L.1 + 1,
    1/t*L.1^2 + 2/t*L.1 + 1/t,
    1/t^2*L.1^3 + 3/t^2*L.1^2 + 3/t^2*L.1 + 1/t^2
]
> pIntegralBasis(I,p:HNF:=true); // In HNF
[
    1,
    L.1,
    1/t*L.1^2 + 2/t*L.1 + 1/t,
    1/t^2*L.1^3 + 3/t^2*L.1^2 + 3/t^2*L.1 + 1/t^2
]
```

The p-integral basis is already the global basis :

```
> Basis(I);
[
    1,
    L.1 + 1,
    1/t*L.1^2 + 2/t*L.1 + 1/t,
    1/t^2*L.1^3 + 3/t^2*L.1^2 + 3/t^2*L.1 + 1/t^2
]
> Basis(I : HNF := true);
[
    1,
    L.1,
    1/t*L.1^2 + 2/t*L.1 + 1/t,
    1/t^2*L.1^3 + 3/t^2*L.1^2 + 3/t^2*L.1 + 1/t^2
]
```

`TwoElement(I): OMIdl -> FldArithElt, FldArithElt`

Given an ideal I in OM representation, return a, b such that $e = a * e_1 + b * e_2$ for some e_1, e_2 for all $e \in I$.

`Norm(I): OMIdl -> RngElt`

Given an ideal I in OM representation, compute the norm of I.

`Valuation(alpha, P : parameters): FldArithElt, OMIdl -> RngIntElt, FldElt`

`Valuation(alpha, P : parameters): FldRatElt, OMIdl -> RngIntElt, FldElt`

`Valuation(alpha, P : parameters): RngIntElt, OMIdl -> RngIntElt, FldElt`

`Valuation(alpha, P : parameters): RngUPolElt, OMIdl -> RngIntElt, FldElt`

```
RED    : BoolElt                    Default: false,
MoreSFL: BoolElt                    Default: false
```

Compute the P-valuation v of α at the prime ideal P.

Setting the parameter `MoreSFL` to `true` selects a single factor lifting algorithm. Setting the parameter `RED` to `true` returns also the class of α in $P^v/P^{(v+1)}$.

`Valuation(I, P): OMIdl, OMIdl -> RngIntElt`

Given ideals I and P in OM representation, return the valuation of I at P.

`a mod P: FldArithElt, OMIdl -> FldArithElt`

`Reduction(a, P): FldArithElt, OMIdl -> FldArithElt`

`Reduction(a, P, m): FldArithElt, OMIdl, RngIntElt -> [FldArithElt]`

Given an element a of the field containing the prime ideal P, which is in OM representation, return a' such that $a = a' + I$ and $a' \in P^0/P$.

If $m > 0$ is given then a sequence of length m of elements in P^0/P is returned representing the local expansion of a at P up to precision m.

`Factorization(I): OMIdl -> SeqEnum`

`Factorisation(I): OMIdl -> SeqEnum`

Given an ideal I in OM representation returns a sequence of tuples of primes P_i and exponents e_i such that $I = \prod_i P_i^{e_i}$.

Example: Creating an OM Ideal (ex-fbb1be)

```
> k := GF(13);
> A<t> := PolynomialRing(k);
> Ax<x> := PolynomialRing(A);
> f := x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3
>         + 3*t^2 + 3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t;
> L := FunctionField(f);
> I := OMRepresentation(L,[1/L.1^2,12]);
> I;
OM ideal of the field Algebraic function field defined over Univariate rational
function field over GF(13) by
x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3 + 3*t^2 +
3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t
generated by [
(t^4 + 4*t^3 + 6*t^2 + 4*t + 1)/t^2*$.1^4 + (t^6 + 6*t^5 + 2*t^4 + 7*t^3 + 2*t^2
+ 5*t + 1)/t^2*$.1^3 + (t^8 + 8*t^7 + 2*t^6 + 4*t^5 + 5*t^4 + 3*t^3 + 7*t +
1)/t^2*$.1^2 + (t^7 + 7*t^6 + 7*t^5 + 5*t^4 + 3*t^3 + 4*t^2 + 6*t + 1)/t^2*$.1 +
(t^8 + 8*t^7 + 2*t^6 + 4*t^5 + 4*t^4 + t^3 + 12*t^2 + 7*t + 1)/t^2,
12
]

> TwoElement(I);
1
(9*t^9 + t^7 + 3*t^6 + 11*t^4 + 2*t^3 + 9*t^2 + 4*t + 1)/t^2*L.1^4 + (4*t^12 +
    8*t^11 + 5*t^10 + 9*t^9 + 2*t^8 + 10*t^7 + 12*t^6 + 6*t^5 + 5*t^4 + 12*t^3 +
    4*t^2 + 5*t + 1)/t^2*L.1^3 + (4*t^11 + 10*t^9 + 11*t^8 + 6*t^7 + 7*t^6 +
    10*t^5 + 5*t^4 + 10*t^3 + 3*t^2 + 7*t + 1)/t^2*L.1^2 + (4*t^12 + 8*t^11 +
    5*t^10 + 2*t^8 + 3*t^7 + 5*t^6 + 8*t^5 + 10*t^4 + 6*t^3 + 7*t^2 + 6*t +
    1)/t^2*L.1 + (4*t^9 + 5*t^8 + 5*t^6 + 11*t^5 + 5*t^4 + 10*t^3 + 3*t^2 + 7*t
    + 1)/t^2
> Norm(I);
1/t^2
```

(continues on next page)

(continued from previous page)

```
> Factorization(I);
[
    <OM prime ideal over   t
    of Algebraic function field defined over Univariate rational function field
    over GF(13) by
    x^5 + (t^2 + 2*t + 1)*x^4 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x^3 + (t^3 +
    3*t^2 + 3*t + 1)*x^2 + (t^4 + 4*t^3 + 6*t^2 + 4*t + 1)*x + t
    having residual degree   1
    and ramification index   1
    Last phi polynomial is   x, -2>
]
> Montes(L, t+1);
> Valuation(I, L`PrimeIdeals[t][1]);
-2
> Valuation(I, L`PrimeIdeals[t+1][1]);
0
```

ResidueField(I): OMIdl -> Fld

Given an ideal I in OM representation returns the field P^0/P.

Degree(I): OMIdl -> RngIntElt

Given a prime ideal I in OM representation, contained in a function field, return the degree of the residue field over the constant field of the function field.

If I is not prime then the degree of the first ideal in the factorization of I is returned.

Example: Degree and Residue Field of OM Ideals (ex-7d7ba3)

```
> F<t> := FunctionField(Rationals());
> P<x> := PolynomialRing(F);
> L<a> := ext<F | x^2 + t>;
> Montes(L, Numerator(t)^2+1);
> L`PrimeIdeals[Numerator(t)^2+1];
[
    OM prime ideal over t^2 + 1
    of Algebraic function field defined over Univariate rational function field
    over Rational Field by
    x^2 + t
    having residual degree   2
    and ramification index   1
    Last phi polynomial is x^2 + t
]
> Degree($1[1]);
4
> ResidueField($2[1]);
Number Field with defining polynomial y^2 + $.1 over its ground field
```

46.21 Divisors in OM representation

OM representations of ideals can be used to construct divisors without computing maximal orders which are required for the divisors of type DivFunElt.

For detailed information please see [Bau14, Bau16b].

46.21.1 Constructing Divisors in OM representation

`OMDivisor(If, Ii): OMIdl, OMIdl -> OMDiv`

Given an ideal If in OM representation whose function field is marked as finite and an ideal Ii in OM representation whose function field is marked as infinite, return the divisor in OM representation whose support is the places associated to the prime ideals which divide either If or Ii with coefficients the negative of valuation of If and Ii at the prime ideals.

`OMDivisor(I): OMIdl -> OMDiv`

Given an ideal I in OM representation return the divisor in OM representation whose support is the places associated to the prime ideals which divide I with coefficients the negative of the valuation of I at the prime ideals.

`OMDivisor(z): FldFunElt -> OMDiv`

Given a non zero element z of a function field, return the principal divisor $\sum_P v_P(z)P$ in OM representation.

`PoleDivisor(a): FldFunElt -> OMDiv`

Given a non zero element a of a function field, return the divisor $\sum_{v_P(a)<0} -v_P(a)P$ in OM representation.

`ZeroDivisor(a): FldFunElt -> OMDiv`

Given a non zero element a of a function field, return the divisor $\sum_{v_P(a)>0} v_P(a)P$ in OM representation.

`OMDivisorOfDegreeOne(F): FldFun -> OMDiv`

Given a global function field F returns a divisor of F having degree 1.

`ReferenceDivisor(F): FldFun -> OMDiv`

Given a function field F, return, if set, the divisor B in OM representation used to reduce a divisor D of F to $\tilde{D}$ such $D = \tilde{D} + kB + (a)$.

`OMDivisor(D): DivFunElt -> OMDiv`

Given a divisor D of a function field of type `DivFunElt` compute an OM representation of D.

Example: Constructing Divisors in OM Representation (ex-42c816)

```
> Fq := GF(13);
> A<t> := PolynomialRing(Fq);
> Ax<x> := PolynomialRing(A);
> f := x^5+(t+1)^16*x^4+ x^2+t^3*x+t^6;
> F := FunctionField(f);
> Montes(F,t);
> Ifin := F`PrimeIdeals[t,1];
> D := OMDivisor(Ifin);
> D;
Free repr. of divisor:  -1*P(t,1)
----------------------------------------
Ideal repr. of divisor:
Finite ideal = P(t,1)
Infinite ideal =
----------------------------------------

> Iinf := PrimesAtInfinity(F)[1];
> D := OMDivisor(Ifin,Iinf);
> D;
Free repr. of divisor:  -1*P(t,1)+( -1*P(1/t,1) )
----------------------------------------
Ideal repr. of divisor:
Finite ideal = P(t,1)
```

(continues on next page)

(continued from previous page)

```
Infinite ideal = P(t,1)
----------------------------------------
> D := OMDivisor(F.1);
> D;
Principal divisor generated by: 12/t^6*$.1^4 + (12*t^16 + 10*t^15 + 10*t^14 +
12*t^13 + 12*t^3 + 10*t^2 + 10*t + 12)/t^6*$.1^3 + 12/t^6*$.1 + 12/t^3
----------------------------------------
Free repr. of divisor:  3*P(t,1)+ 3*P(t,2)+( 5*P(1/t,1)+ 5*P(1/t,2)+
-16*P(1/t,3) )
----------------------------------------
Ideal repr. of divisor:
Finite ideal = P(t,1)^-3*P(t,2)^-3
Infinite ideal = P(t,1)^-5*P(t,2)^-5*P(t,3)^16
----------------------------------------

> E := PoleDivisor(F.1);
> E;
Free repr. of divisor:  16*P(1/t,3)
----------------------------------------
Ideal repr. of divisor:
Finite ideal =
Infinite ideal = P(t,3)^-16
----------------------------------------

> C := ZeroDivisor(F.1);
> C;
Free repr. of divisor:  3*P(t,1)+ 3*P(t,2)+( 5*P(1/t,1)+ 5*P(1/t,2) )
----------------------------------------
Ideal repr. of divisor:
Finite ideal = P(t,1)^-3*P(t,2)^-3
Infinite ideal = P(t,1)^-5*P(t,2)^-5
----------------------------------------

> E  := Divisor(F.1);
> D  := OMDivisor(E);
> D;
Free repr. of divisor:  3*P(t,1)+ 3*P(t,2)+( 5*P(1/t,1)+ 5*P(1/t,2)+
-16*P(1/t,3) )
----------------------------------------
Ideal repr. of divisor:
Finite ideal = P(t,1)^-3*P(t,2)^-3
Infinite ideal = P(t,1)^-5*P(t,2)^-5*P(t,3)^16
----------------------------------------
```

46.21.2 Arithmetic with Divisors in OM representation

`k * D: RngIntElt, OMDiv -> OMDiv`

Given an integer k and a divisor D in OM representation return the scalar multiple kD.

`D1 + D2: OMDiv, OMDiv -> OMDiv`

Given divisors $D1$ and $D2$ in OM representation return their sum also in OM representation.

`D1 - D2: OMDiv, OMDiv -> OMDiv`

Given divisors $D1$ and $D2$ in OM representation return their difference also in OM representation.

`GCD(D1, D2): OMDiv, OMDiv -> OMDiv`

`Gcd(D1, D2): OMDiv, OMDiv -> OMDiv`

`GreatestCommonDivisor(D1, D2): OMDiv, OMDiv -> OMDiv`

Given divisors $D1$ and $D2$ in OM representation return their GCD in OM representation.

Example: Arithmetic of OM Divisors `(ex-31985e)`

```
> Fq := GF(13);
> A<t> := PolynomialRing(Fq);
> Ax<x> := PolynomialRing(A);
> f := x^5+(t+1)^16*x^4+ x^2+t^3*x+t^6;
> F := FunctionField(f);
> Montes(F,t);
> Ifin := F`PrimeIdeals[t,1];
> D := OMDivisor(Ifin);
> E := PoleDivisor(F.1);
> C := ZeroDivisor(F.1);
> D + E;
Free repr. of divisor:  -1*P(t,1)+( 16*P(1/t,3) )
----------------------------------------
Ideal repr. of divisor:
Finite ideal = P(t,1)
Infinite ideal = P(t,3)^-16
----------------------------------------

> D - C;
Free repr. of divisor:  -4*P(t,1)+ -3*P(t,2)+( -5*P(1/t,1)+ -5*P(1/t,2) )
----------------------------------------
Ideal repr. of divisor:
Finite ideal = P(t,1)^4*P(t,2)^3
Infinite ideal = P(t,1)^5*P(t,2)^5
----------------------------------------

> 5*C;
Free repr. of divisor:  15*P(t,1)+ 15*P(t,2)+( 25*P(1/t,1)+ 25*P(1/t,2) )
----------------------------------------
Ideal repr. of divisor:
Finite ideal = P(t,1)^-15*P(t,2)^-15
Infinite ideal = P(t,1)^-25*P(t,2)^-25
----------------------------------------
> GCD(D, C);
Free repr. of divisor:  -1*P(t,1)
----------------------------------------
Ideal repr. of divisor:
Finite ideal = P(t,1)
Infinite ideal =
----------------------------------------

> GCD(D, E);
Free repr. of divisor:
----------------------------------------
Ideal repr. of divisor:
Finite ideal =
Infinite ideal =
----------------------------------------
```

46.21.3 Predicates on Divisors in OM representation

IsEffective(D): OMDiv -> BoolElt

IsPositive(D): OMDiv -> BoolElt

Given a divisor D in OM representation, return whether the valuation of D at every prime in its support is non negative.

IsPrincipal(D): OMDiv -> BoolElt

Given a divisor D in OM representation, return whether D was constructed as the divisor of an element.

D1 **eq** D2: OMDiv, OMDiv -> BoolElt

Given divisors $D1$ and $D2$ in OM representation, return whether they are equal.

Example: Predicates on OM Divisors (ex-c8e530)

Continuing with the previous example :

```
> IsPrincipal(OMDivisor(F.1));
true
> IsPositive(OMDivisor(F.1));
false
> IsPositive(ZeroDivisor(F.1));
true
> ZeroDivisor(F.1) eq OMDivisor(F.1);
false
```

46.21.4 Other Operations on Divisors in OM representation

Divisor(D): OMDiv -> DivFunElt

Given a divisor D in OM representation return a divisor of type **DivFunElt**.

Valuation(D, P): OMDiv, OMIdl -> RngIntElt

Given a divisor D in OM representation and a prime ideal P in OM representation, return the valuation of D at P.

Support(D): OMDiv -> [OMIdl], [RngIntElt]

Given a divisor D in OM representation, return a sequence of primes at which D has non zero valuation and a sequence of the valuations that D has at those primes.

Height(D): OMDiv -> RngIntElt

CanonicalHeight(D): OMDiv -> RngIntElt

Given a divisor D in OM representation, return the sum of the products of the absolute values of the valuations returned by **Support** with the degree of the primes in the **Support** of D and the maximum degree of the primes in the support of D.

Degree(D): OMDiv -> RngIntElt

Given a divisor D in OM representation, return the sum of the products of the valuations returned by **Support** with the degrees of the primes in the support of D.

Basis(D): OMDiv -> SeqEnum

Given a divisor D in OM representation, return a basis of the Riemann–Roch space of D.

Dimension(D): OMDiv -> RngIntElt

Given a divisor D in OM representation, return the dimension of its Riemann–Roch space.

ReducedBasis(D): OMDiv -> SeqEnum

Given a divisor D in OM representation returns a reduced basis for the lattice corresponding to D, that is, the lattice given by its finite ideal.

SemiReducedBasis(D): OMDiv -> SeqEnum

Given a divisor D in OM representation returns a semi-reduced basis for the lattice corresponding to D, that is, the lattice given by its finite ideal.

SuccessiveMinima(D): OMDiv -> SeqEnum

Given a divisor D in OM representation returns the successive minima of the lattice given by the finite ideal of D.

ApproximatedSuccessiveMinima(D): OMDiv -> SeqEnum

Given a divisor D in OM representation returns the ceiling of the successive minima of the lattice given by the finite ideal of D.

Example: Operations on OM Divisors (ex-e8bf40)

Continuing with the previous example :

```
> Montes(F, t);
> Ifin := F`PrimeIdeals[t,1];
> D := OMDivisor(Ifin);
> D;
Free repr. of divisor:  -1*P(t,1)
----------------------------------------
Ideal repr. of divisor:
Finite ideal = P(t,1)
Infinite ideal =
----------------------------------------

> Support(D);
[
    OM prime ideal over   t
    of Algebraic function field defined over Univariate rational function field
    over GF(13) by
    x^5 + (t^16 + 3*t^15 + 3*t^14 + t^13 + t^3 + 3*t^2 + 3*t + 1)*x^4 + x^2 +
    t^3*x + t^6
    having residual degree   1
    and ramification index   1
    Last phi polynomial is x + 10*t^3
]
[ -1 ]
> Height(D);
1 1
> Degree(D);
-1
> Basis(OMDivisor(F.1));
[
    12/t^6*F.1^4 + (12*t^16 + 10*t^15 + 10*t^14 + 12*t^13 + 12*t^3 + 10*t^2 +
        10*t + 12)/t^6*F.1^3 + 12/t^6*F.1 + 12/t^3
]
> Dimension(OMDivisor(F.1));
1
> ReducedBasis(OMDivisor(F.1));
[
    12/t^6*F.1^4 + (12*t^16 + 10*t^15 + 10*t^14 + 12*t^13 + 12*t^3 + 10*t^2 +
        10*t + 12)/t^6*F.1^3 + 12/t^6*F.1 + 12/t^3,
    1/t^3*F.1^3 + (t^16 + 3*t^15 + 3*t^14 + t^13 + t^3 + 3*t^2 + 3*t +
        1)/t^3*F.1^2 + 1/t^3,
    12*F.1^2 + (12*t^16 + 10*t^15 + 10*t^14 + 12*t^13 + 12*t^3 + 10*t^2 + 10*t +
        12)*F.1,
    F.1 + t^16 + 3*t^15 + 3*t^14 + t^13 + t^3 + 3*t^2 + 3*t + 1,
```

(continues on next page)

(continued from previous page)

```
    F.1^2 + (t^16 + 3*t^15 + 3*t^14 + t^13 + t^3)*F.1 + 10*t^18 + t^17 + 7*t^16
        + 11*t^15 + 7*t^14 + 12*t^13 + 10*t^5 + t^4 + 7*t^3 + 11*t^2 + 7*t
]
> SemiReducedBasis(OMDivisor(F.1));
[
    12/t^6*F.1^4 + (12*t^16 + 10*t^15 + 10*t^14 + 12*t^13 + 12*t^3 + 10*t^2 +
        10*t + 12)/t^6*F.1^3 + 12/t^6*F.1 + 12/t^3,
    1/t^3*F.1^3 + (t^16 + 3*t^15 + 3*t^14 + t^13 + t^3 + 3*t^2 + 3*t +
        1)/t^3*F.1^2 + 1/t^3,
    12*F.1^2 + (12*t^16 + 10*t^15 + 10*t^14 + 12*t^13 + 12*t^3 + 10*t^2 + 10*t +
        12)*F.1,
    F.1 + t^16 + 3*t^15 + 3*t^14 + t^13 + t^3 + 3*t^2 + 3*t + 1,
    F.1^2 + (t^16 + 3*t^15 + 3*t^14 + t^13 + t^3)*F.1 + 10*t^18 + t^17 + 7*t^16
        + 11*t^15 + 7*t^14 + 12*t^13 + 10*t^5 + t^4 + 7*t^3 + 11*t^2 + 7*t
]
> SuccessiveMinima(OMDivisor(F.1));
[ 0, 11/2, 11, 27/2, 16 ]
> ApproximatedSuccessiveMinima(OMDivisor(F.1));
[ 0, 6, 11, 14, 16 ]
```

46.22 Bibliography

- [Bau14] Jens–Dietrich Bauch. *Lattices over Polynomial Rings and Applications to Function Fields*. PhD thesis, Universitat Autònoma de Barcelona, 2014.
- [Bau16a] Jens-Dietrich Bauch. Computation of integral bases. *Journal of Number Theory*, 165:382–407, 2016. URL: https://www.sciencedirect.com/science/article/pii/S0022314X16000743, doi:https://doi.org/10.1016/j.jnt.2016.01.011.
- [Bau16b] Jens-Dietrich Bauch. Lattices over Polynomial Rings and Applications to Function Fields. https://arxiv.org/abs/1601.01361, 2016. arXiv:1601.01361.
- [Bla05] I. Blake, G. Serrousi and N. Smart, editor. *Advances in Elliptic Curve Cryptography*. Volume 317 of LMS LNS. Cambridge University Press, Cambridge, 2005.
- [BL94c] Johannes A. Buchmann and Hendrik W. Lenstra jr. Approximating rings of integers in number fields. *J. Théor. Nombres Bordx.*, 6(2):221–260, 1994.
- [CHM98] John H. Conway, Alexander Hulpke, and John McKay. On transitive permutation groups. *LMS Journal of Computation and Mathematics*, 1:1–8, 1998.
- [Els14] A.-S. Elsenhans. A Note on Short Cosets. *Experimental Mathematics*, 23:411–413, 2014. doi:10.1080/10586458.2014.922908.
- [FK14] C. Fieker and J. Klüners. Computation of Galois Groups of Rational Polynomials. *London Mathematical Society Journal of Computation and Mathematics*, 17(1):141 – 158, 2014. http://arxiv.org/abs/1211.3588.
- [Gei03] Katharina Geißler. *Berechnung von Galoisgruppen über Zahl- und Funktionenkörpern*. PhD Thesis, TU-Berlin, 2003. available at http://www.math.tu-berlin.de/ kant/publications/diss/geissler.pdf.
- [GK00] Katharina Geißler and Jürgen Klüners. Galois Group computation for Rational Polynomials. *J. Symbolic Comp.*, 30(6):653–674, 2000.
- [GMN13] Jordi Guàrdia, Jesús Montes, and Enric Nart. A New Computational Approach to Ideal Theory in Number Fields. *Foundations of Computational Mathematics*, 13(5):729–762, Oct 2013. URL: https://doi.org/10.1007/s10208-012-9137-5, doi:10.1007/s10208-012-9137-5.
- [GNP12] Jordi Guàrdia, Enric Nart, and Sebastian Pauli. Single-factor lifting and factorization of polynomials over local fields. *Journal of Symbolic Computation*, 47(11):1318

– 1346, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0747717112000363, doi:https://doi.org/10.1016/j.jsc.2012.03.001.

- [Hes] F. Heß. "An algorithm for computing isomorphisms of algebraic function fields." 263–271.
- [Hes99] Florian Heß. *Zur Divisorenklassengruppenberechnung in globalen Funktionenkörpern*. Dissertation, Technische Universität Berlin, 1999. http://www.staff.uni-oldenburg.de/florian.hess/publications/diss_FH.pdf.
- [Klu02] Jürgen Klüners. Algorithms for Function Fields. *Exp. Math.*, pages 171–181, 2002.
- [KS21] David Krumm and Nicole Sutherland. Galois groups over rational function fields and Explicit Hilbert Irreducibility. *Journal of Symbolic Computation*, 103:108 – 126, 2021. URL: http://www.sciencedirect.com/science/article/pii/S0747717119301476, doi:https://doi.org/10.1016/j.jsc.2019.11.003.
- [PGS00] F. Heß P. Gaudry and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *J. Cryptology*, 15(1):19–46, 2000.
- [SM85] Leonhard H. Soicher and John McKay. Computing Galois Groups over the rationals. *J. Number Th.*, 20:273–281, 1985.
- [Sta18] Hayden D. Stainsby. Triangular bases of integral closures. *J. Symbolic Comput.*, 87:140–175, 2018. URL: https://doi.org/10.1016/j.jsc.2017.07.012, doi:10.1016/j.jsc.2017.07.012.
- [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. ISBN 3-540-56489-6.
- [Sut12] Nicole Sutherland. Efficient Computation of Maximal Orders of Radical (including Kummer) Extensions. *Journal of Symbolic Computation*, 47(5):552–567, 2012. doi:10.1016/j.jsc.2011.12.053.
- [Sut13] Nicole Sutherland. Efficient Computation of Maximal Orders of Artin–Schreier Extensions. *Journal of Symbolic Computation*, 53(1):26–39, 2013. URL: http://dx.doi.org/10.1016/j.jsc.2012.11.001, doi:10.1016/j.jsc.2012.11.001.
- [Sut15] Nicole Sutherland. Computing Galois Groups of Polynomials (especially over Function Fields of Prime Characteristic). *Journal of Symbolic Computation*, 71:73–97, 2015. doi:10.1016/j.jsc.2014.09.043.
- [Sut18] Nicole Sutherland. Galois Groups in Magma. *Computeralgebra Rundbrief*, 62:, 2018.
- [vHKN11] M. van Hoeij, J. Klüners, and A. Novocin. Generating Subfields. In Anton Leykin, editor, *Proceedings ISSAC 2011*. 2011.

Chapter 47

Class Field Theory for Global Function Fields

Global function fields admit a class field theory in the same way as number fields do (Chapter *Class Field Theory*). 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 *Analytic Theory*.

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.

47.1 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): DivFunElt -> GrpAb, Map`

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): DivFunElt -> GrpAb, Map`

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): DivFunElt, DivFunElt -> GrpAbElt

RayClassGroupDiscLog(y, D): PlcFunElt, DivFunElt -> GrpAbElt

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: Ray Class and Ray Residue Groups (ex-d6bc20)

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,
```

(continues on next page)

(continued from previous page)

```
    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.

47.2 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): DivFunElt, GrpAb -> FldFunAb`

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): FldFunG -> FldFunAb`

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): FldFun, PlcFunElt -> FldFunAb`

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): FldFunAb -> FldFun`

```
WithAut   : BoolElt                    Default: false
ClassField: Verbose                    Default: 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): FldFunAb -> RngFunOrd`

`MaximalOrderInfinite(A): FldFunAb -> RngFunOrd`

Compute a finite or an infinite maximal order of the function field of the abelian extension A.

This computes the maximal orders of each function field defined by each of the defining polynomials of the function field of A then combines them and finishes off the computation by using the algorithm of [BL94c] on the result. As the cyclic components are essentially Kummer, Artin–Schreier, or Artin–Schreier—Witt extensions the algorithms in [Sut12, Sut13] and [Sut16] are used to compute the maximal orders of the component function fields.

Example: Creating Class Fields of a Function Field `(ex-df5eee)`

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. To finish we compute maximal orders of A.

```
> MaximalOrderFinite(A);
Maximal Order of Algebraic function field defined over K by
$.1^5 + (t*K.1^2 + (w*t^2 + w^2)*K.1 + (w*t^2 + t + w))*$.1^3 + ((t^2 + w)*K.1^2
    + w^2*t^3*K.1 + (t^4 + t^2 + w^2))*$.1 + (t^4 + w^2*t^2 + w*t + w)*K.1^2 +
    (w*t^5 + w*t^4 + w*t^3)*K.1 + w^2*t^5 + t^4 + w*t^3 + w^2*t + 1 over Maximal
    Equation Order of K over Univariate Polynomial Ring in t over GF(2^2)
> MaximalOrderInfinite(A);
Maximal Order of Algebraic function field defined over K by
$.1^5 + (w^2*K.1^2 + (w^2*t + 1)*K.1 + (t^2 + w*t + 1))*$.1^3 + ((w^2*t^2 +
    w^2)*K.1^2 + t*K.1 + (t^4 + 1))*$.1 + (w*t^3 + w^2*t^2 + t)*K.1^2 + (w^2*t^4
    + t^2 + 1)*K.1 + w*t^4 + t^3 + t + w^2 over Maximal Order of K over
    Valuation ring of Univariate rational function field over GF(2^2) with
    generator 1/t
> DiscriminantDivisor(A);
4*(t^2 + t + w, K.1 + w^2*t + 1) + 4*(t^2 + w^2*t + w^2, K.1 + w*t + w)
> $1 eq Divisor(Discriminant($3)) + Divisor(Discriminant($2));
true
```

Example: Degree 4 Class Fields with Many Places (ex-5a3adf)

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;
7168
> 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
```

(continues on next page)

(continued from previous page)

```
> 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. But we will compute the maximal orders.

```
> MaximalOrderFinite(s4[15][3]);
Maximal Order of E over Maximal Equation Order of K over Univariate Polynomial
Ring in t over GF(2^2)
> MaximalOrderInfinite(s4[15][3]);
Maximal Order of E over Maximal Order of K over Valuation ring of Univariate
rational function field over GF(2^2) with generator 1/t
```

47.3 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. Therefore none of the functions listed here will compute a set of defining equations. They can therefore be used on very large fields.

`Conductor(m): DivFunElt -> DivFunElt`

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): DivFunElt, GrpAb -> DivFunElt`

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): FldFunAb -> DivFunElt`

For an abelian extension A of global function fields, compute its conductor, ie. the conductor of the norm group of A.

`DiscriminantDivisor(m, U): DivFunElt, GrpAb -> DivFunElt`

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): FldFunAb -> DivFunElt`

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): DivFunElt -> RngIntElt`

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): DivFunElt, GrpAb -> RngIntElt`

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): FldFunAb -> RngIntElt`

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): DivFunElt, GrpAb -> RngIntElt`

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): FldFunAb -> RngIntElt`

The genus of the abelian extension A of a global function field.

`DecompositionType(m, U, p): DivFunElt, GrpAb, PlcFunElt -> [<f,e>]`

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): FldFunAb, PlcFunElt -> [<f,e>]`

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): DivFunElt, GrpAb -> RngIntElt`

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): FldFunAb -> RngIntElt`

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): FldFunAb -> RngIntElt`

For an abelian extension A of global function fields, return the degree of A over its base ring.

`BaseField(A): FldFunAb -> FldFunG`

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: FldFunAb, FldFunAb -> BoolElt`

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: FldFunAb, FldFunAb -> BoolElt`

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: FldFunAb, FldFunAb -> FldFunAb`

Compute the intersection of two abelian extensions of the same base field as an abelian extension.

`A * B: FldFunAb, FldFunAb -> FldFunAb`

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.

47.4 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): Fld, RngIntElt -> RngWitt

Creates the ring of Witt vectors of length n where F must be a field of positive characteristic.

W ! a: RngWitt, . -> RngWittElt

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): RngWitt -> Fld

BaseField(W): RngWitt -> Fld

The field of coefficients of the Witt ring W.

Length(W): RngWitt -> RngIntElt

The length (dimension) of the elements of the Witt ring W.

Eltseq(a): RngWittElt -> [FldElt]

The list of coefficients of the Witt vector a.

One(W): RngWitt -> RngWittElt

Zero(W): RngWitt -> RngWittElt

The one and zero in the Witt ring W.

W . 1: RngWitt, RngIntElt -> RngWittElt

The first non-trivial basis element of the Witt ring.

x **in** W: ., RngWitt -> BoolElt

a **eq** b: RngWittElt, RngWittElt -> BoolElt

a **eq** b: RngWitt, RngWitt -> BoolElt

a - b: RngWittElt, RngIntElt -> RngWittElt

a - b: RngIntElt, RngWittElt -> RngWittElt

a - b: RngWittElt, RngWittElt -> RngWittElt

- a: RngWittElt -> RngWittElt

a + b: RngWittElt, RngIntElt -> RngWittElt

a + b: RngIntElt, RngWittElt -> RngWittElt

a + b: RngWittElt, RngWittElt -> RngWittElt

a * b: RngWittElt, RngIntElt -> RngWittElt

a * b: RngIntElt, RngWittElt -> RngWittElt

a * b: RngWittElt, RngWittElt -> RngWittElt

a ^ n: RngWittElt, RngIntElt -> RngWittElt

FrobeniusMap(W): RngWitt -> Map

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): RngWittElt -> RngWittElt

Computes the image of the Witt vector e under the Frobenius map.

`VerschiebungMap(W): RngWitt -> Map`

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): RngWittElt -> RngWittElt`

Computes the image of the Witt vector e under the verschiebung map.

`Random(W): RngWitt -> RngWittElt`

For finite Witt rings, i.e. Witt rings defined over finite fields, return a random element.

`Random(W, n): RngWitt, RngIntElt -> RngWittElt`

For Witt rings where the base field admits a random function with size restriction n.

`TeichmuellerSystem(R): Any -> [RngLocElt]`

`TeichmuellerSystem(R): Rng -> [RngLocElt]`

A Teichmüller 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): RngWitt -> RngLoc, Map`

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): RngWitt -> Map`

Returns the map $x \mapsto F(x) - x$ where F is the Frobenius map of the Witt ring W.

`ArtinSchreierImage(e): RngWittElt -> RngWittElt`

This function computes the image of the Witt vector e under the Artin-Schreier map.

`FunctionField(e): RngWittElt -> FldFun, Map`

```
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.

`MaximalOrderFinite(u): RngWittElt -> RngFunOrd`

`MaximalOrderInfinite(u): RngWittElt -> RngFunOrd`

Given a Witt vector u defined over a function field F return the finite, respectively infinite, maximal order of the degree p^n Artin–Schreier–Witt extension of F defined by u. This uses the algorithm described in [Sut16].

`SMaximalOrder(u, S): RngWittElt, [PlcFunElt] -> RngFunOrd`

Given a Witt vector u of length n defined over a function field F of characteristic p and a set S of places of F all contained in the same maximal order $\mathbb{Z}_F$ of F, return the minimal S-maximal order of the degree p^n Artin–Schreier–Witt extension E of F defined by u containing the equation order of E which extends $\mathbb{Z}_F$. This uses the algorithm described in [Sut16].

`MaximalOrders(u): RngWittElt -> RngFunOrd, RngFunOrd`

Given a Witt vector u over a function field F compute the finite and infinite maximal orders of the Artin–Schreier–Witt extension $F(\wp^{-}1(u))$ where $\wp$ is the Artin–Schreier operator. This uses the algorithm described in [Sut16].

47.5 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.

47.5.1 Creation of Twisted Polynomial Rings

`TwistedPolynomials(R): Rng -> RngUPolTwst`

`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.

47.5.2 Operations with the Ring of Twisted Polynomials

`One(R): RngUPolTwst -> RngUPolTwstElt`

For a ring R of twisted polynomials return the polynomial representing 1.

`Zero(R): RngUPolTwst -> RngUPolTwstElt`

For a ring R of twisted polynomials return the polynomial representing 0.

`R eq S: RngUPolTwst, RngUPolTwst -> BoolElt`

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): RngUPolTwst -> Rng`

The coefficient ring of the ring R of twisted polynomials.

`R . i: RngUPolTwst, RngIntElt -> RngUPolTwstElt`

For a ring of twisted polynomials R and an integer i which should be 1, return the transcendental element of R.

47.5.3 Creation of Twisted Polynomials

Apart from creation from polynomials directly, twisted polynomial can also be created by specifying some finite $\mathbb{F}_q$-vector space where the corresponding additive polynomial will have its roots.

`AdditivePolynomialFromRoots(x, P): RngElt, PlcFunElt -> RngUPolTwstElt`

```
InfBound: RngIntElt                    Default: 5
Map     : Map                          Default: id
Class   : DivFunElt                    Default: 0
Limit   : RngIntElt                    Default: Infinity()
Scale   : RngElt                       Default: false
```

An additive polynomial is characterized by the fact that its set of zeros forms an $\mathbb{F}_q$ vector space. Conversely, given an $\mathbb{F}_p$-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 $L(\texttt{InfBound}P)$, 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 $x^{\texttt{Limit}}$. If `Class` is set to a non-zero divisor, instead of $L(nP)$, the Riemann-Roch space $L(nP+\texttt{Class})$ is used. If `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): RngUPolTwst, RngIntElt -> RngUPolTwstElt`

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: Twisted and Additive Polynomials `(ex-f3126f)`

```
> 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 ]
```

(continues on next page)

(continued from previous page)

```
> 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)
```

47.5.4 Operations with Twisted Polynomials

`A + B: RngIntElt, RngUPolTwstElt -> RngUPolTwstElt`

`A + B: RngUPolTwstElt, RngIntElt -> RngUPolTwstElt`

`A + B: RngUPolTwstElt, RngUPolTwstElt -> RngUPolTwstElt`

`A - B: RngIntElt, RngUPolTwstElt -> RngUPolTwstElt`

`A - B: RngUPolTwstElt, RngIntElt -> RngUPolTwstElt`

`A - B: RngUPolTwstElt, RngUPolTwstElt -> RngUPolTwstElt`

`- A: RngUPolTwstElt -> RngUPolTwstElt`

`A * B: RngUPolTwstElt, RngUPolTwstElt -> RngUPolTwstElt`

`A ^ n: RngUPolTwstElt, RngIntElt -> RngUPolTwstElt`

`A eq B: RngUPolTwstElt, RngUPolTwstElt -> BoolElt`

`IsZero(A): RngUPolTwstElt -> BoolElt`

`LeadingCoefficient(F): RngUPolTwstElt -> RngElt`

For a twisted polynomial F, return the leading coefficient as an element of the coefficient ring.

`ConstantCoefficient(F): RngUPolTwstElt -> RngElt`

For a twisted polynomial F, return the constant coefficient as an element of the coefficient ring.

`Degree(F): RngUPolTwstElt -> RngIntElt`

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): RngUPolTwstElt, RngUPolTwstElt -> RngUPolTwstElt, RngUPolTwstElt`

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): RngUPolTwstElt, RngUPolTwstElt -> RngUPolTwstElt`

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 $F = f_1 H$ and $G = f_2 H$ for some twisted polynomials f_1 and f_2 and such that H is monic of maximal degree.

`BaseRing(F): RngUPolTwstElt -> Rng`

For a twisted polynomial F, return the coefficient ring of F, ie. the ring where all the coefficients of F are from.

`Polynomial(G): RngUPolTwstElt -> RngUPolElt`

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): RngUPolTwstElt, RngElt -> RngElt

For a twisted polynomial F, return the result of the evaluation of the corresponding additive polynomial at the point x.

SpecialEvaluate(F, x): RngUPolElt, Any -> RngElt

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): RngUPolTwstElt -> [RngElt]

For a twisted polynomial F, return a sequence containing its coefficients.

47.6 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): RngUPolTwst, RngUPolElt -> RngUPolTwstElt

CarlitzModule(R, x): RngUPolTwst, FldFunRatUElt -> RngUPolTwstElt

For a rational function field $k = \mathbb{F}_q(t)$ 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 $\mathbb{F}_q[t]$ is freely generated by t, this defines a homomorphism (a representation) of $\mathbb{F}_q[t]$ into the twisted polynomials over k.

Example: Abelian Extensions from the Carlitz Module (ex-15c3a6)

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 ]
```

(continues on next page)

(continued from previous page)

```
> RamificationIndex(a[1]);
15
```

So, this shows that the field has the ramification behaviour 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 $\mathbb{F}_q[t]/p$ 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): FldFun, PlcFunElt -> RngUPolTwstElt

For F a global function field and p a place, compute an algebraic description of "the" Drinfeld module 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 having 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. Furthermore, 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): RngUPolTwstElt, RngElt, PlcFunElt -> RngUPolTwstElt

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: Drinfeld Module on an Elliptic Curve (ex-cecb57)

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
```

(continues on next page)

(continued from previous page)

```
+ 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): RngElt, PlcFunElt -> RngUPolTwstElt

```
InfBound: RngIntElt                    Default: 5
Map     : Map                          Default: id
Class   : DivFunElt                    Default: 0
Limit   : RngIntElt                    Default: Infinity()
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): RngElt, PlcFunElt -> RngElt`

```
InfBound: RngIntElt                    Default: 5
Map     : Map                          Default: id
Class   : DivFunElt                    Default: 0
Limit   : RngIntElt                    Default: Infinity()
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): RngUPolTwstElt -> BoolElt, RngUPolTwstElt, RngElt`

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): RngUPolTwstElt -> BoolElt, RngUPolTwstElt, RngElt`

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): RngUPolTwstElt, PlcFunElt -> RngUPolTwstElt`

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.

47.7 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): DivFunElt, [<PlcFunElt, FldFunElt>] -> FldFunElt`

```
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.

`StrongApproximation(S, Z, V): [PlcFunElt], [FldFunGElt], [RngIntElt] -> FldFunElt`

```
Strict: BoolElt                 Default: false
```

Given a sequence S of either finite or infinite places of a function field, a sequence Z of elements of a function field and a sequence V of integers, return an element z such that $z - Z[i]$ has valuation at least $V[i]$ at $S[i]$ and positive valuation at all other places of same finiteness as those in S which do not appear in S. If the parameter `Strict` is set to `true` then z will be computed such that $v_{S[i]}(z - Z[i]) = V[i]$.

`ChineseRemainderTheorem(S, Z, V): [PlcFunElt], [FldFunGElt], [RngIntElt] -> FldFunElt`
`CRT(S, Z, V): [PlcFunElt], [FldFunGElt], [RngIntElt] -> FldFunElt`

```
IntegralOutside: BoolElt                Default: false
```

Given a sequence S of either finite or infinite places of a function field, a sequence Z of elements of a function field and a sequence V of integers, return an element z such that $z - Z[i]$ has valuation at least $V[i]$ at $S[i]$. If the parameter `IntegralOutside` is set to `true` then z will be computed such that it is integral at all other places of same finiteness as those in S which do not appear in S.

Example: Strong Approximation (ex-8bc2d0)

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): DivFunElt -> DivFunElt, RngIntElt`

```
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): FldFun -> DivFunElt, GrpAb`

```
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): FldFunElt, PlcFunElt -> RngElt`

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): FldFun, PlcFunElt -> FldFun`

Given a global function field F and a place p, return a new function field G that is $\mathbb{F}_q$-isomorphic to F and has p as the only infinite place.

`ArtinSchreierReduction(u, P): FldFunGElt, PlcFunElt -> RngIntElt, FldFunElt`

Return the valuation of $u-(z^p-z)$ at P and an element z such that this valuation is either positive or not congruent to $0 \mod p$ where p is the characteristic of the field of u.

47.8 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): FldFun -> PlcEnum

```
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): PlcFunElt -> PlcEnum

```
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): FldFun, [RngIntElt] -> PlcEnum

```
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): PlcEnum -> PlcEnum

Copies the environment and the current state of the enumeration process for places R.

PlaceEnumPosition(R): PlcEnum -> [RngIntElt]

Returns a list of integers that acts as an index to the places as enumerated by the environment R.

PlaceEnumNext(R): PlcEnum -> PlcFunElt

Returns the "next" place of the process R.

PlaceEnumCurrent(R): PlcEnum -> PlcFunElt

Returns the current place pointed to by the environment R, i.e. the last place returned by `PlaceEnumNext`.

47.9 Bibliography

- [BL94c] Johannes A. Buchmann and Hendrik W. Lenstra jr. Approximating rings of integers in number fields. *J. Théor. Nombres Bordx.*, 6(2):221–260, 1994.
- [HPP97] Florian Heß, Sebastian Pauli, and Michael E. Pohst. On the computation of the multiplicative group of residue class rings. *Math. Comp.*, 1997.
- [Sut16] N. Sutherland. Efficient Computation of Maximal Orders in Artin–Schreier–Witt Extensions. *Journal of Symbolic Computation*, 77:189–216, 2016. doi:10.1016/j.jsc.2016.01.008.
- [Sut12] Nicole Sutherland. Efficient Computation of Maximal Orders of Radical (including Kummer) Extensions. *Journal of Symbolic Computation*, 47(5):552–567, 2012. doi:10.1016/j.jsc.2011.12.053.

- [Sut13] Nicole Sutherland. Efficient Computation of Maximal Orders of Artin–Schreier Extensions. *Journal of Symbolic Computation*, 53(1):26–39, 2013. URL: http://dx.doi.org/10.1016/j.jsc.2012.11.001, doi:10.1016/j.jsc.2012.11.001.

Chapter 48

Artin Representations

48.1 Overview

An *Artin representation* is a complex representation of $\mathrm{Gal}(\bar{\mathbb{Q}}/\mathbb{Q})$ that factors through some finite quotient $\mathrm{Gal}(F/\mathbb{Q})$. In Magma, Artin representations are represented as characters of $\mathrm{Gal}(F/\mathbb{Q})$, not the actual modules. They are allowed to be virtual, except in the L-function machinery (see Chapter L-series).

48.2 Constructing Artin Representations

`ArtinRepresentations(K): FldNum -> SeqEnum`

```
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.

If K is given as a relative extension E/F (that is, `BaseField(K)` is not $\mathbb{Q}$), the representations returned are those of $\mathrm{Gal}(\bar{F}/F)$ that factor through the normal closure of E over F, and `LSeries` returns the corresponding Artin L-series over F. To obtain the representations of $\mathrm{Gal}(\bar{\mathbb{Q}}/\mathbb{Q})$ instead, apply `ArtinRepresentations` to `AbsoluteField(K)`.

Such relative representations support the operations that derive one Artin representation from another: sums, differences, and tensor products of representations attached to the same field, `Decomposition` and `RationalDecomposition` return relative representations over the same base field, and `Kernel` returns the smallest extension of the base field through which the representation factors. `IsRamified`, `IsWildlyRamified`, `Minimize`, `OptimizedRepresentation`, `DirichletCharacter`, and `HeckeCharacter` are not implemented over a base field other than $\mathbb{Q}$, and raise an error for a relative representation rather than answering over $\mathbb{Q}$.

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 $\mathbb{Q}$. (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: FldNum, AlgChtrElt -> ArtRep`

`K !! ch: FldNum, SeqEnum -> ArtRep`

Writing F for the normal closure of $K/\mathbb{Q}$, this function converts an abstract group character of $\mathrm{Gal}(F/\mathbb{Q})$ or the sequence of its trace values into an Artin representation.

`PermutationCharacter(K): FldNum -> ArtRep`

Construct the permutation representation A of the absolute Galois group of $\mathbb{Q}$ on the embeddings of K into $\mathbb{C}$. This is an Artin representation of $\mathrm{Gal}(F/\mathbb{Q})$ of dimension $[K:\mathbb{Q}]$, where F is the normal closure of K, and it is the same as the permutation representation of $\mathrm{Gal}(F/\mathbb{Q})$ on the cosets of $\mathrm{Gal}(F/K)$.

`Determinant(A): ArtRep -> ArtRep`

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): ArtRep, FldNum -> ArtRep, BoolElt`

`K !! A: FldNum, ArtRep -> ArtRep, BoolElt`

```
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: Artin Representations of a Quadratic Field `(ex-81e6bd)`

A quadratic field K has two irreducible Artin representations the factor through $\mathrm{Gal}(K/\mathbb{Q})$, the trivial one and the quadratic character of K:

```
> K<i> := QuadraticField(-1);
> triv, sign := Explode(ArtinRepresentations(K));
> sign;
Artin representation C2: (1,-1) of Q(sqrt(-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 $\mathrm{Gal}(K/\mathbb{Q})$ is their sum:

```
> PermutationCharacter(K);
Artin representation C2: (2,0) of Q(sqrt(-1))
> $1 eq triv+sign;
true
```

Next, let $L = K(\sqrt{-2-i})$, presented as a relative extension of K. As L is defined over K, `ArtinRepresentations` returns the representations of $\mathrm{Gal}(\bar{K}/K)$, here the trivial character and the quadratic character of $\mathrm{Gal}(L/K)$:

```
> P<x> := PolynomialRing(K);
> L := ext<K | x^2 + i + 2>;
> reps := ArtinRepresentations(L);
> [Dimension(A): A in reps];
[ 1, 1 ]
> BaseField(reps[2]) eq K;
true
```

The normal closure F of L over $\mathbb{Q}$ has $\mathrm{Gal}(F/\mathbb{Q}) = D_4$, the dihedral group of order 8. Passing to the absolute field instead gives the Artin representations of $\mathrm{Gal}(\bar{\mathbb{Q}}/\mathbb{Q})$:

```
> Labs := AbsoluteField(L);
> GroupName(GaloisGroup(Labs));
D4
> [Dimension(A): A in ArtinRepresentations(Labs)];
[ 1, 1, 1, 1, 2 ]
```

We use `ChangeField` to lift Artin representations from $\mathrm{Gal}(K/\mathbb{Q})$ to $\mathrm{Gal}(F/\mathbb{Q})$, and check that it is still the same as an Artin representation.

```
> A := ChangeField(sign,L);
> A;
Artin representation D4: (1,1,-1,1,-1) of ext<Q(sqrt(-1))|x^2+i+2>
> A eq sign;
true
```

48.3 Basic Invariants

`Field(A): ArtRep -> FldNum`

Number field K such that A factors through the Galois group of the normal closure of K.

`BaseField(A): ArtRep -> Fld`

The field F over which A is a representation of $\mathrm{Gal}(\bar{F}/F)$. This is $\mathbb{Q}$ unless A was constructed as a relative Artin representation, in which case it is the base field of the defining relative extension.

`Degree(A): ArtRep -> RngIntElt`

`Dimension(A): ArtRep -> RngIntElt`

Degree (=dimension) of an Artin representation A.

`Group(A): ArtRep -> GrpPerm`

The Galois group of the field through which A factors.

`Character(A): ArtRep -> AlgChtrElt`

Character of an Artin representation A, represented as a complex-valued character of `Group(A)`.

`Conductor(A): ArtRep -> RngIntElt`

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): ArtRep -> SeqEnum[Tup]`

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). If A is a relative Artin representation, the decomposition is taken in the Galois group over its base field, and the A_i are relative Artin representations over that same base field.

`RationalDecomposition(A): ArtRep -> SeqEnum[Tup]`

Decompose an Artin representation A into constituents with rational-valued characters, each corresponding to a Galois conjugacy class of irreducibles. Returns a sequence of tuples `[...<A_i, n_i>...]` with A_i having rational-valued character and n_i its exponent in A (nonzero but possibly negative). As with `Decomposition`, a relative Artin representation decomposes into relative Artin representations over the same base field.

Example: Decomposing Artin Representations (ex-12f333)

We give an example to show the difference between these decompositions.

```
> K := NumberField(PolynomialWithGaloisGroup(12,15));
> DefiningPolynomial(K);
x^12 - 24*x^10 + 216*x^8 - 896*x^6 + 1680*x^4 - 1152*x^2 + 48
> GroupName(GaloisGroup(K));
C3:D4
> c := PermutationCharacter(K);
> Decomposition(c);
[ <Artin representation C3:D4: (1,1,1,1,1,1,1,1,1,1) of K, 1>,
  <Artin representation C3:D4: (1,1,1,-1,1,-1,1,1,1) of K, 1>,
  <Artin representation C3:D4: (2,2,2,0,-1,0,-1,-1,-1) of K, 2>,
  <Artin representation C3:D4: (2,-2,0,0,2,0,0,0,-2) of K, 1>,
  <Artin representation C3:D4: (2,-2,0,0,-1,0,-1-2*J,1+2*J,1) of K, 1>,
  <Artin representation C3:D4: (2,-2,0,0,-1,0,1+2*J,-1-2*J,1) of K, 1> ]
> RationalDecomposition(c);
[ <Artin representation C3:D4: (1,1,1,1,1,1,1,1,1,1) of K, 1>,
  <Artin representation C3:D4: (1,1,1,-1,1,-1,1,1,1) of K, 1>,
  <Artin representation C3:D4: (2,2,2,0,-1,0,-1,-1,-1) of K, 2>,
  <Artin representation C3:D4: (2,-2,0,0,2,0,0,0,-2) of K, 1>,
  <Artin representation C3:D4: (4,-4,0,0,-2,0,0,0,2) of K, 1>];
```

`DefiningPolynomial(A): ArtRep -> RngUPolElt`

Returns the polynomial whose roots `Group(A)` permutes.

`Minimize(A): ArtRep -> ArtRep`

```
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`. Raises an error when A is a relative Artin representation, as this is not implemented over a base field other than $\mathbb{Q}$.

`OptimizedRepresentation(A): ArtRep -> ArtRep`

Returns the same Artin representation, but over an isomorphic version of the field that has had its representation optimized. Raises an error when A is a relative Artin representation, as this is not implemented over a base field other than $\mathbb{Q}$.

`Kernel(A): ArtRep -> FldNum`

Smallest Galois extension K of the rationals through which A factors. Note that this field may be enormous and incomputable. For a relative Artin representation this is instead the smallest

extension of the base field, Galois over it, through which A factors. In both cases the result is returned as a number field over $\mathbb{Q}$.

Example: Minimising an Artin Representation (ex-9129cc)

We take an S_4-extension of $\mathbb{Q}$ 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 S3: (2,0,-1) of ext<Q|x^3+8*x+81>
> Kernel(B);
Number Field with defining polynomial x^6 + 48*x^4 + 576*x^2 + 179195
   over the Rational Field
```

`IsIrreducible(A): ArtRep -> BoolElt`

Return `true` iff a given Artin representation is irreducible as a complex representation.

`IsRamified(A, p): ArtRep, RngIntElt -> BoolElt`

Return `true` iff a given Artin representation is ramified at p. Raises an error when A is a relative Artin representation, as this is not implemented over a base field other than $\mathbb{Q}$.

`IsWildlyRamified(A, p): ArtRep, RngIntElt -> BoolElt`

Return `true` iff a given Artin representation is wildly ramified at p. Raises an error when A is a relative Artin representation, as this is not implemented over a base field other than $\mathbb{Q}$.

`EulerFactor(A, p): ArtRep, RngIntElt -> RngUPolElt`

```
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.

`EpsilonFactor(A): ArtRep -> FldComElt`

Global epsilon-factor $\epsilon(A)$ of an Artin representation. Currently only implemented in a few basic cases, and raises an error otherwise. See Example Example: Local and Global Epsilon Factors for Dirichlet Characters.

`RootNumber(A): ArtRep -> FldComElt`

Global root number $\epsilon(A)/|\epsilon(A)|$ of an Artin representation. Currently only implemented in a few basic cases, and raises an error otherwise. See Example Example: Local and Global Epsilon Factors for Dirichlet Characters.

`EpsilonFactor(A, p): ArtRep, RngIntElt -> FldComElt`

Local epsilon-factor $\epsilon(A)$ of an Artin representation at p. Currently only implemented in a few basic cases, and raises an error otherwise. See Example Example: Local and Global Epsilon Factors for Dirichlet Characters.

`RootNumber(A, p): ArtRep, RngIntElt -> FldComElt`

Local root number $\epsilon_p(A)/|\epsilon_p(A)|$ of an Artin representation at p. Currently only implemented in a few basic cases, and raises an error otherwise. See Example Example: Local and Global Epsilon Factors for Dirichlet Characters.

`EpsilonFactor(A, infty): ArtRep, Infty -> FldComElt`

`RootNumber(A, infty): ArtRep, Infty -> FldComElt`

Local root number $w_\infty(A)$ of an Artin representation at infinity. See Example Example: Local and Global Epsilon Factors for Dirichlet Characters.

Example: Invariants of Artin Representations (ex-2bf31b)

Here are the invariants of Artin representations that factor through the splitting field of $x^4 - 3$, a D_4-extension of $\mathbb{Q}$.

```
> R<x> := PolynomialRing(Rationals());
> K := NumberField(x^4-3);
> A := ArtinRepresentations(K);
> Degree(Kernel(A[5]),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, 12, 3, 4, 576 ]
> [IsRamified(a,3): a in A];
[ false, true, true, false, true ]
> [IsWildlyRamified(a,3): a in A];
[ false, false, false, false, false ]
> EulerFactor(A[5],5);
x^2 + 1
> EpsilonFactor(A[5],3);
-3
```

`DirichletCharacter(A): ArtRep -> GrpDrchElt`

Convert a one-dimensional Artin representation to a Dirichlet character. Raises an error when A is a relative Artin representation, as this is not implemented over a base field other than $\mathbb{Q}$.

`HeckeCharacter(A): ArtRep -> GrpHeckeElt`

Convert a one-dimensional Artin representation A to a Hecke character. This is more natural than the previous, as in general Hecke characters will have L-functions matching that of the Artin representation, while Dirichlet characters only necessarily have L-functions when defined over the rationals. Raises an error when A is a relative Artin representation, as this is not implemented over a base field other than $\mathbb{Q}$.

`ArtinRepresentation(ch): GrpDrchElt -> ArtRep`

```
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: One-Dimensional Artin Representations (ex-db71ab)

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];
```

(continues on next page)

(continued from previous page)

```
[ 1, 2, 4, 4, 2, 2, 2, 1, 1, 2, 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 C4: (1,-1,-I,I) of ext<Q|x^4+x^3-54*x^2-54*x+551>
> Factorization(Discriminant(Integers(Field($1))));
[ <5, 3>, <43, 2> ]
```

48.4 Arithmetic

`A1 + A2: ArtRep, ArtRep -> ArtRep`

Direct sum of two Artin representations

`A1 - A2: ArtRep, ArtRep -> ArtRep`

Direct difference of two Artin representations

`A1 * A2: ArtRep, ArtRep -> ArtRep`

Tensor product of two Artin representations

`A1 eq A2: ArtRep, ArtRep -> BoolElt`

Returns `true` iff the two Artin representations are equal

`A1 ne A2: ArtRep, ArtRep -> BoolElt`

Returns `true` iff the two Artin representations are not equal

Sums, differences, and tensor products of relative Artin representations attached to the same number field are relative Artin representations over the same base field. Arithmetic between representations attached to different number fields, which passes to the compositum, is available only over $\mathbb{Q}$.

Example: Arithmetic of Artin Representations (ex-ba9cf8)

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 S3: (1,1,1) of ext<Q|x^3-2>, conductor 1
> rho;
Artin representation S3: (2,0,-1) of ext<Q|x^3-2>, conductor 108
> triv+rho;
Artin representation S3: (3,1,0) of ext<Q|x^3-2>, conductor 108
> sign*rho eq rho;
true
```

Example: Artin Representations over Different Fields (ex-401593)

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));
```

(continues on next page)

(continued from previous page)

```
> twist := sign1*sign2;
> Field(twist);
Number Field with defining polynomial x^4 - 10*x^2 + 1 over the Rational Field
> sign3 := Minimize(twist);
> sign3;
Artin representation C2: (1,-1) of ext<Q|x^2-6>
> sign1*sign2*sign3 eq triv1;
true
```

48.5 Implementation Notes

The algorithms for recognizing Frobenius elements in Galois groups are described in [DD13]. They rely on the cycle type identification, Serre's trick for alternating groups, and the general machinery from [DD13]. 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.

48.6 Bibliography

- [DD13] T. Dokchitser and V. Dokchitser. Identifying Frobenius elements in Galois groups. *Algebra Number Theory*, 7(6):1325–1352, 2013.

References

[Dim] "Dimino's Algorithm." 13–23.

[AK99] Vincenzo Acciaro and Jürgen Klüners. Computing Automorphisms of Abelian Number Fields. *Math. Comp.*, 68(227):1179–1186, 1999.

[ACvHS17] Emilio Allem, Juliane Capaverde, Mark van Hoeij, and Jonas Szutkoski. Functional Decomposition using Principal Subfields. In *ISSAC '17, July 25-28, 2017, Kaiserslautern, Germany*. ACM, 2017. accepted. doi:doi.org/10.1145/3087604.3087608.

[Bai96] Georg Baier. Zum Round 4 Algorithmus. Master's thesis, Technische Universität Berlin, 1996. http://www.math.tu-berlin.de/∼kant/publications/diplom/baier.ps.gz.

[Bau14] Jens–Dietrich Bauch. *Lattices over Polynomial Rings and Applications to Function Fields*. PhD thesis, Universitat Autònoma de Barcelona, 2014.

[Bau16a] Jens-Dietrich Bauch. Computation of integral bases. *Journal of Number Theory*, 165:382–407, 2016. URL: https://www.sciencedirect.com/science/article/pii/S0022314X16000743, doi:https://doi.org/10.1016/j.jnt.2016.01.011.

[Bau16b] Jens-Dietrich Bauch. Lattices over Polynomial Rings and Applications to Function Fields. https://arxiv.org/abs/1601.01361, 2016. arXiv:1601.01361.

[BL94a] 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.

[BF15] Karim Belabas and Eduardo Friedman. Computing the residue of the Dedekind zeta function. *Math. Comp.*, 84(291):357–369, 2015. URL: https://doi.org/10.1090/S0025-5718-2014-02843-3, doi:10.1090/S0025-5718-2014-02843-3.

[Ber05] Daniel J. Bernstein. Factoring into coprimes in essentially linear time. *J. of Algorithms*, 54(1):1–30, 2005.

[BDE+] N. Berry, A. Dubickas, N. Elkies, B. Poonen, C. Smyth. "The conjugate dimension of algebraic numbers." Quart. J. Math., 55, 237–252.

[Bia] J. Biasse. "Number field sieve to compute Class groups."

[BH96] Yuri Bilu and Guillaume Hanrot. Solving Thue Equations of High Degree. *J. Number Th.*, 60:373–392, 1996.

[Bla05] I. Blake, G. Serrousi and N. Smart, editor. *Advances in Elliptic Curve Cryptography*. Volume 317 of LMS LNS. Cambridge University Press, Cambridge, 2005.

[BL94c] Johannes A. Buchmann and Hendrik W. Lenstra jr. Approximating rings of integers in number fields. *J. Théor. Nombres Bordx.*, 6(2):221–260, 1994.

[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.

[CDO] H. Cohen, F. Diaz y Diaz, M. Olivier. "Computing Ray Class Groups, Conductors and Discriminants." 52–59.

[CDO98] Henri Cohen, Francisco Diaz y Diaz, and Michel Olivier. Computing Ray Class Groups, Conductors and Discriminants. *Math. Comp.*, 67(222):773–795, 1998.

[CHM98] John H. Conway, Alexander Hulpke, and John McKay. On transitive permutation groups. *LMS Journal of Computation and Mathematics*, 1:1–8, 1998.

[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.

[Der94] Harm Derksen. An algorithm to compute generalized Padé-Hermite Forms. Technical Report 9403, Department of Mathemtaics, Catholic University Nijmegen, jan 1994.

[DD13] T. Dokchitser and V. Dokchitser. Identifying Frobenius elements in Galois groups. *Algebra Number Theory*, 7(6):1325–1352, 2013.

[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, 289–290. Linz, 1985. Springer.

[Els12] A.-S. Elsenhans. Invariants for the computation of intransitive and transitive Galois groups. *Journal of Symbolic Computation*, 47:315–326, 2012.

[Els14] A.-S. Elsenhans. A Note on Short Cosets. *Experimental Mathematics*, 23:411–413, 2014. doi:10.1080/10586458.2014.922908.

[Els16] Andreas-Stephan Elsenhans. Improved methods for the construction of relative invariants for permutation groups. To appear in Journal of Symbolic Computation, Available online since 5 February 2016, 2016. doi:10.1016/j.jsc.2016.02.005.

[FJP97] C. Fieker, A. Jurk, and M. Pohst. On solving relative norm equations in algebraic number fields. *Math. Comput.*, 66(217):399–410, 1997.

[FK14] C. Fieker and J. Klüners. Computation of Galois Groups of Rational Polynomials. *London Mathematical Society Journal of Computation and Mathematics*, 17(1):141 – 158, 2014. http://arxiv.org/abs/1211.3588.

[Fie97] Claus Fieker. *Über relative Normgleichungen in algebraischen Zahlkörpern*. Dissertation, Technische Universität Berlin, 1997. http://www.math.tu-berlin.de/∼kant/publications/diss/diss_CF.ps.gz.

[Fie00] Claus Fieker. Computing Class Fields via the Artin Map. *Math. Comput.*, 70(235):1293–1303, 2000.

[FF] C. Fieker, C. Friedrichs. "On reconstruction of algebraic numbers." 285–296.

[Fin84] Ulrich Fincke. *Ein Ellipsoidverfahren zur Lösung von Normgleichungen in algebraischen Zahlkörpern*. Dissertation, Heinrich-Heine-Universität Düsseldorf, 1984.

[Fri97] Carsten Friedrichs. Berechnung relativer Ganzheitsbasen mit dem Round-2-Algorithmus. Master's thesis, Technische Universität Berlin, 1997. 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. http://www.math.tu-berlin.de/∼kant/publications/diss/diss_fried.pdf.gz.

[GPP93] Istvan Gaál, Attila Petho, 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 Petho, 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.

[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.

[Gei03] Katharina Geißler. *Berechnung von Galoisgruppen über Zahl- und Funktionenkörpern*. PhD Thesis, TU-Berlin, 2003. available at http://www.math.tu-berlin.de/ kant/publications/diss/geissler.pdf.

[GK00] Katharina Geißler and Jürgen Klüners. Galois Group computation for Rational Polynomials. *J. Symbolic Comp.*, 30(6):653–674, 2000.

[GMN13] Jordi Guàrdia, Jesús Montes, and Enric Nart. A New Computational Approach to Ideal Theory in Number Fields. *Foundations of Computational Mathematics*, 13(5):729–762, Oct 2013. URL: https://doi.org/10.1007/s10208-012-9137-5, doi:10.1007/s10208-012-9137-5.

[GNP12] Jordi Guàrdia, Enric Nart, and Sebastian Pauli. Single-factor lifting and factorization of polynomials over local fields. *Journal of Symbolic Computation*, 47(11):1318 – 1346, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0747717112000363, doi:https://doi.org/10.1016/j.jsc.2012.03.001.

[Hes] F. Heß. "An algorithm for computing isomorphisms of algebraic function fields." 263–271.

[Hes96] Florian Heß. Zur Klassengruppenberechnung in algebraischen Zahlkörpern. Master's thesis, Technische Universität Berlin, 1996. http://www.math.tu-berlin.de/∼kant/publications/diplom/hess.ps.gz.

[Hes99] Florian Heß. *Zur Divisorenklassengruppenberechnung in globalen Funktionenkörpern*. Dissertation, Technische Universität Berlin, 1999. http://www.staff.uni-oldenburg.de/florian.hess/publications/diss_FH.pdf.

[HPP97] Florian Heß, Sebastian Pauli, and Michael E. Pohst. On the computation of the multiplicative group of residue class rings. *Math. Comp.*, 1997.

[HS20] Tommy Hofmann and Carlo Sircana. On the computation of overorders. *International Journal of Number Theory*, 16(04):857–879, 2020. URL: https://doi.org/10.1142/S179304212050044X, arXiv:https://doi.org/10.1142/S179304212050044X, doi:10.1142/S179304212050044X.

[Jur93] Andreas Jurk. *Über die Berechnung von Lösungen relativer Normgleichungen in algebraischen Zahlkörpern*. Dissertation, Heinrich-Heine-Universität Düsseldorf, 1993.

[Klu95] Jürgen Klüners. Über die Berechnung von Teilkörpern algebraischer Zahlkörper. Master's thesis, Technische Universität Berlin, 1995. http://www.math.tu-berlin.de/∼kant/publications/diplom/klueners.ps.gz.

[Klu97] Jürgen Klüners. *Über die Berechnung von Automorphismen und Teilkörpern algebraischer Zahlkörper*. Dissertation, Technische Universität Berlin, 1997. http://www.math.tu-berlin.de/∼kant/publications/diss/diss_jk.ps.gz.

[Klu98] Jürgen Klüners. On computing subfields. A detailed description of the algorithm. *J. Theor. Nombres Bordx.*, 2(10):243–271, 1998.

[Klu02] Jürgen Klüners. Algorithms for Function Fields. *Exp. Math.*, pages 171–181, 2002.

[KP97] Jürgen Klüners and Michael E. Pohst. On Computing Subfields. *J. Symbolic Comp.*, 24(3):385–397, 1997.

[KS21] David Krumm and Nicole Sutherland. Galois groups over rational function fields and Explicit Hilbert Irreducibility. *Journal of Symbolic Computation*, 103:108 – 126, 2021. URL: http://www.sciencedirect.com/science/article/pii/S0747717119301476, doi:https://doi.org/10.1016/j.jsc.2019.11.003.

[Mar20] Stefano Marseglia. Computing the ideal class monoid of an order. *J. Lond. Math. Soc. (2)*, 101(3):984–1007, 2020. URL: https://doi.org/10.1112/jlms.12294, doi:10.1112/jlms.12294.

[Mar24] Stefano Marseglia. Cohen-Macaulay type of orders, generators and ideal classes. *J. Algebra*, 658:247–276, 2024. URL: https://doi.org/10.1016/j.jalgebra.2024.05.051, doi:10.1016/j.jalgebra.2024.05.051.

[Mar25] Stefano Marseglia. Local isomorphism classes of fractional ideals of orders in étale algebras. *J. Algebra*, 673:77–102, 2025. URL: https://doi.org/10.1016/j.jalgebra.2025.02.030, doi:10.1016/j.jalgebra.2025.02.030.

[Mul25] Erik Mulder. Fast square-free decomposition of integers using class groups. *Res. Number Theory*, 11(1):Paper No. 9, 21, 2025. URL: https://doi.org/10.1007/s40993-024-00585-8, doi:10.1007/s40993-024-00585-8.

[PGS00] F. Heß P. Gaudry and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *J. Cryptology*, 15(1):19–46, 2000.

[Pau96] Sebastian Pauli. Zur Berechnung von Strahlklassengruppen. Master's thesis, Technische Universität Berlin, 1996. http://www.math.tu-berlin.de/ kant/publications/diplom/pauli.ps.gz.

[Pau01b] Sebastian Pauli. Factoring polynomials over local fields. *Journal of Symbolic Computation*, 32(5):533–547, 2001.

[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.

[Poh] M. E. Pohst. "Computational Aspects of Kummer Theory." 259–272.

[PZ89] Michael E. Pohst and Hans Zassenhaus. *Algorithmic Algebraic Number Theory*. Encyclopaedia of mathematics and its applications. Cambridge University Press, Cambridge, 1989.

[Roh] D. E. Rohrlich. "Root Numbers." In Arithmetic L-functions. 18, 353–448.

[Sim02] Denis Simon. Solving norm equations in relative number fields using S-units. *Math. Comput.*, 71(239):1287–1305, 2002.

[SM85] Leonhard H. Soicher and John McKay. Computing Galois Groups over the rationals. *J. Number Th.*, 20:273–281, 1985.

[Sta18] Hayden D. Stainsby. Triangular bases of integral closures. *J. Symbolic Comput.*, 87:140–175, 2018. URL: https://doi.org/10.1016/j.jsc.2017.07.012, doi:10.1016/j.jsc.2017.07.012.

[Sta73] Richard P. Stauduhar. The determination of Galois Groups. *Math. Comp.*, 27:981–996, 1973.

[Ste] A. Steel. "A new scheme for computing with algebraically closed fields." 491–505.

[Ste10] Allan K. Steel. Computing with algebraically closed fields. *J. Symb. Comput.*, 45(3):342–372, March 2010. URL: http://dx.doi.org/10.1016/j.jsc.2009.09.005, doi:10.1016/j.jsc.2009.09.005.

[Sti93] Henning Stichtenoth. *Algebraic function fields and codes*. Springer-Verlag, Berlin, 1993. ISBN 3-540-56489-6.

[Sut16] N. Sutherland. Efficient Computation of Maximal Orders in Artin–Schreier–Witt Extensions. *Journal of Symbolic Computation*, 77:189–216, 2016. doi:10.1016/j.jsc.2016.01.008.

[Sut12] Nicole Sutherland. Efficient Computation of Maximal Orders of Radical (including Kummer) Extensions. *Journal of Symbolic Computation*, 47(5):552–567, 2012. doi:10.1016/j.jsc.2011.12.053.

[Sut13] Nicole Sutherland. Efficient Computation of Maximal Orders of Artin–Schreier Extensions. *Journal of Symbolic Computation*, 53(1):26–39, 2013. URL: http://dx.doi.org/10.1016/j.jsc.2012.11.001, doi:10.1016/j.jsc.2012.11.001.

[Sut15] Nicole Sutherland. Computing Galois Groups of Polynomials (especially over Function Fields of Prime Characteristic). *Journal of Symbolic Computation*, 71:73–97, 2015. doi:10.1016/j.jsc.2014.09.043.

[Sut18] Nicole Sutherland. Galois Groups in Magma. *Computeralgebra Rundbrief*, 62:, 2018.

[Tra76] Barry M. Trager. Algebraic factoring and rational function integration. In R.D. Jenks, editor, *Proc. SYMSAC '76*, 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.

[vHKN11] M. van Hoeij, J. Klüners, and A. Novocin. Generating Subfields. In Anton Leykin, editor, *Proceedings ISSAC 2011*. 2011.

[Wat11] M. Watkins. Computing with Hecke Grössencharacters. *Publications mathématiques de Besançon*, 2011/1:119–135, 2011.

[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. 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.

[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. http://www.math.tu-berlin.de/∼kant, 2000.

Index

Symbols

+ a (*operation*), 22, 83, 254, 266, 321
+ w (*operation*), 83
D (*operation*), 32, 365
L (*operation*), 166
W (*operation*), 223
icm (*operation*), 225
&* L (*operation*), 115
&* seq (*operation*), 205
&+ seq (*operation*), 205, 211
&meet S (*operation*), 117, 166
- A (*operation*), 392
- D (*operation*), 346
- P (*operation*), 343
- a (*operation*), 22, 83, 126, 254, 266, 321, 340, 388
- d (*operation*), 29, 130
- p (*operation*), 29, 130
- w (*operation*), 83
- x (*operation*), 153, 205, 360
1 * P (*operation*), 345
3-Selmer Group of a Real Quadratic Field (*example*), 178

A

A ! a (*operation*), 247
A ! x (*operation*), 204
A * B (*operation*), 184, 387, 392
a * b (*operation*), 22, 83, 126, 169, 254, 267, 321, 340, 388
a *:= b (*operation*), 254
A + B (*operation*), 392
a + b (*operation*), 22, 83, 126, 254, 266, 321, 340, 388
a +:= b (*operation*), 254
A . i (*operation*), 207, 248
a / b (*operation*), 22, 83, 126, 254, 267, 321, 340
A - B (*operation*), 392
a - b (*operation*), 22, 83, 126, 254, 266, 321, 340, 388
a -:= b (*operation*), 254
a ^ k (*operation*), 22, 83, 254, 267, 321
A ^ n (*operation*), 392
a ^ n (*operation*), 126, 169, 340, 388
a div b (*operation*), 39, 126, 321
A eq B (*operation*), 184, 254, 387, 392
a eq b (*operation*), 22, 85, 126, 169, 255, 267, 322, 340, 388
a in A (*operation*), 255
a in F (*operation*), 23, 85, 267, 322
a in FF (*operation*), 322
a in I (*operation*), 135, 370
a in O (*operation*), 322
A meet B (*operation*), 184, 387
a mod b (*operation*), 39, 126
a mod I (*operation*), 117, 126, 321, 339
a mod P (*operation*), 137, 372
A ne B (*operation*), 254
a ne b (*operation*), 22, 85, 126, 255, 267, 322, 340
a notin A (*operation*), 255
a notin F (*operation*), 267, 322
a notin FF (*operation*), 322
a notin O (*operation*), 322
A subset B (*operation*), 184, 387
A1 * A2 (*operation*), 408
A1 + A2 (*operation*), 408
A1 - A2 (*operation*), 408
A1 eq A2 (*operation*), 203, 408
A1 ne A2 (*operation*), 408
A`Components (*attribute*), 196
A`DefiningGroup (*attribute*), 197
A`IsAbelian (*attribute*), 197
A`IsCentral (*attribute*), 197
A`IsNormal (*attribute*), 197
A`NormGroup (*attribute*), 197
Abelian Extensions from the Carlitz Module (*example*), 393
AbelianExtension(chi) (*function*), 187
AbelianExtension(D, U) (*function*), 383
AbelianExtension(I) (*function*), 181
AbelianExtension(I, P) (*function*), 183
AbelianExtension(K) (*function*), 183
AbelianExtension(M) (*function*), 183
AbelianExtension(m) (*function*), 180
AbelianExtension(m, I) (*function*), 180
AbelianExtension(m, I, T) (*function*), 180
AbelianExtension(psi) (*function*), 187
AbelianpExtension(m, p) (*function*), 181
AbelianSubfield(A, U) (*function*), 186
AbsoluteAffineAlgebra(A) (*function*), 257
AbsoluteBasis(A) (*function*), 207
AbsoluteBasis(K) (*function*), 17, 78
AbsoluteCharacteristicPolynomial(a) (*function*), 24, 90
AbsoluteCoordinates(seq, basis) (*function*), 207
AbsoluteCoordinates(seq, I) (*function*), 210
AbsoluteCoordinates(seq, O) (*function*), 208

AbsoluteCoordinates(x) (*function*), 204
AbsoluteCoordinates(x, I) (*function*), 210
AbsoluteCoordinates(x, S) (*function*), 204
AbsoluteDegree(A) (*function*), 188
AbsoluteDegree(F) (*function*), 14, 75, 291
AbsoluteDegree(O) (*function*), 75, 291
AbsoluteDimension(A) (*function*), 203
AbsoluteDiscriminant(A) (*function*), 188
AbsoluteDiscriminant(K) (*function*), 15, 75
AbsoluteDiscriminant(O) (*function*), 75, 292
AbsoluteField(F) (*function*), 10, 67
AbsoluteFunctionField(F) (*function*), 289
AbsoluteGaloisGroup(A) (*function*), 191
AbsoluteInertiaDegree(I) (*function*), 110
AbsoluteInertiaIndex(I) (*function*), 110
AbsoluteLogarithmicHeight(a) (*function*), 88
AbsoluteMinimalPolynomial(a) (*function*), 25, 90, 324
AbsoluteMinimalPolynomial(x) (*function*), 206
AbsoluteNorm(a) (*function*), 24, 89
AbsoluteNorm(I) (*function*), 109
AbsoluteNorm(x) (*function*), 215
AbsoluteOrder(O) (*function*), 67, 289
AbsolutePolynomial(A) (*procedure*), 257
AbsoluteQuotientRing(A) (*function*), 257
AbsoluteRamificationDegree(I) (*function*), 110
AbsoluteRamificationIndex(I) (*function*), 110
AbsoluteRepresentationMatrix(a) (*function*), 25, 90
AbsoluteTrace(a) (*function*), 24, 89
AbsoluteTrace(x) (*function*), 215
AbsoluteValues(a) (*function*), 87
Absolutize(A) (*procedure*), 257
AddGenerator(G, x) (*function*), 200
AdditivePolynomialFromRoots(x, P) (*function*), 390
AffineAlgebra(A) (*function*), 253
Algebra(I) (*function*), 210
Algebra(K, J) (*function*), 13, 73
Algebra(K, J, S) (*function*), 13, 73
Algebra(S) (*function*), 207
Algebra(x) (*function*), 204
Algebraic Relation for Sine and Cosine (*example*), 352
AlgebraicClosure() (*function*), 247
AlgebraicClosure(K) (*function*), 247
AlgebraicToAnalytic(F, p) (*function*), 397
AllCMTypes(A) (*function*), 227
Alternative Discriminant Function (*example*), 88
AnalyticDrinfeldModule(F, p) (*function*), 394
AnalyticModule(x, p) (*function*), 397
Annihilator(a) (*function*), 127
Approximants of Sine, Cosine, and Exponential (*example*), 278
ApproximatedSuccessiveMinima(D) (*function*), 378
Arithmetic of Artin Representations (*example*), 408
Arithmetic of OM Divisors (*example*), 376
Arithmetic of OM Ideals (*example*), 134, 369
Array(W) (*function*), 223
Artin Representations of a Quadratic Field (*example*), 403
Artin Representations over Different Fields (*example*), 408
ArtinMap(A) (*function*), 189
ArtinRepresentation(ch) (*function*), 407
ArtinRepresentations(K) (*function*), 402
ArtinSchreierExtension(c,a,b) (*function*), 363
ArtinSchreierImage(e) (*function*), 389
ArtinSchreierMap(W) (*function*), 389
ArtinSchreierReduction(u, P) (*function*), 399
AsExtensionOf(O, P) (*function*), 55
AsExtensionOf(O1, O2) (*function*), 286
AssignNamePrefix(A, S) (*procedure*), 247
AssignNames(~a, s) (*procedure*), 281
AssignNames(~D, s) (*procedure*), 345
AssignNames(~F, s) (*procedure*), 67, 264, 281
AssignNames(~F, [s]) (*procedure*), 36
AssignNames(~K, s) (*procedure*), 9, 66
AssignNames(~O, [s]) (*procedure*), 36
AssignNames(~P, s) (*procedure*), 341
AssociatedPrimitiveCharacter(chi) (*function*), 230
AssociatedPrimitiveCharacter(psi) (*function*), 230
AssociatedPrimitiveGrossencharacter(psi) (*function*), 237
Attributes of a Ray Class Field (*example*), 197
Automorphism(a) (*function*), 169
AutomorphismGroup(A) (*function*), 190
AutomorphismGroup(F) (*function*), 142
AutomorphismGroup(K) (*function*), 26, 308
AutomorphismGroup(K, F) (*function*), 142
AutomorphismGroup(K, k) (*function*), 305
AutomorphismGroup(K,f) (*function*), 309
Automorphisms of a Degree 7 Extension (*example*), 305
Automorphisms of a Quartic Field (*example*), 142
Automorphisms(F) (*function*), 141
Automorphisms(K) (*function*), 308
Automorphisms(K, k) (*function*), 305
Automorphisms(K,p1,p2) (*function*), 308
a[i] (*literal*), 26, 92

B

BachBound(K) (*function*), 95
BachBound(O) (*function*), 95
BaseField(A) (*function*), 188, 203, 253, 387, 404
BaseField(a) (*function*), 169
BaseField(D) (*function*), 365
BaseField(F) (*function*), 10, 67, 288

BaseField(FF) (*function*), 288
BaseField(W) (*function*), 388
BaseRing(A) (*function*), 188
BaseRing(F) (*function*), 265, 288, 392
BaseRing(FF) (*function*), 288
BaseRing(O) (*function*), 67, 288
BaseRing(R) (*function*), 153, 390
BaseRing(W) (*function*), 388
Bases of an Order and Its Fraction Field (*example*), 54
Basic Access to the Number Field Database (*example*), 32
Basic Character Operations (*example*), 233
Basis and Fundamental Units of a Maximal Order (*example*), 315
Basis and Integral Basis (*example*), 17, 79
Basis of an Ideal (*example*), 112
Basis over Different Rings (*example*), 16, 78
Basis(A) (*function*), 207
Basis(D : parameters) (*function*), 350
Basis(D) (*function*), 377
Basis(F) (*function*), 16, 77, 292
Basis(F, R) (*function*), 16, 77
Basis(I) (*function*), 111, 136, 335, 370
Basis(I, R) (*function*), 111, 335
Basis(O) (*function*), 77, 292
Basis(O, R) (*function*), 77, 292
BasisMatrix(I) (*function*), 111, 335
BasisMatrix(O) (*function*), 78, 292
BasisOfDifferentialsFirstKind(F) (*function*), 358
BasisOfHolomorphicDifferentials(F) (*function*), 358
BiquadraticResidueSymbol(a, b) (*function*), 40
Bottom(L) (*function*), 166
Bound(I, B) (*function*), 155

C

c * I (*operation*), 331
c / I (*operation*), 331
CanComputeEmbedding(F, L) (*function*), 71
CanComputeEmbedding(O1, O2) (*function*), 71, 289
CanNormalize(F) (*function*), 397
Canonical Grossencharacter of Q(sqrt(-23)) (*example*), 238
CanonicalDivisor(F) (*function*), 345
CanonicalHeight(D) (*function*), 377
CanSignNormalize(F) (*function*), 397
CarlitzModule(R, x) (*function*), 393
Cartier(b) (*function*), 361
Cartier(b, r) (*function*), 361
CartierRepresentation(F) (*function*), 362
CartierRepresentation(F, r) (*function*), 362
Category(A) (*function*), 252
Category(a) (*function*), 22, 83, 254, 320
Category(F) (*function*), 66, 287
Category(I) (*function*), 330
Category(K) (*function*), 9
Category(O) (*function*), 66, 287
Category(P) (*function*), 342
Category(R) (*function*), 265
Category(w) (*function*), 83
Central Character of a Dirichlet Character (*example*), 235
CentralCharacter(chi) (*function*), 235
CentralCharacter(GR) (*function*), 238
CentralCharacter(psi) (*function*), 235
Centre(A) (*function*), 252
Centre(F) (*function*), 11, 71
Centre(O) (*function*), 71
ChangeField(A,K) (*function*), 403
ChangeModel(F, p) (*function*), 399
ChangePrecision(~PHI, prec) (*procedure*), 227
ChangePrecision(~R, k) (*procedure*), 128
ChangePrecision(PHI0, prec) (*function*), 227
ChangeRing(a, K) (*function*), 169
Character(A) (*function*), 404
Characteristic(A) (*function*), 254
Characteristic(F) (*function*), 14, 75, 265, 291
Characteristic(O) (*function*), 75, 291
CharacteristicPolynomial(a) (*function*), 24, 90, 323
CharacteristicPolynomial(a, R) (*function*), 24, 90, 323
Chinese Remainder Theorem Maps (*example*), 218
ChineseRemainderTheorem(I, J, a, b) (*function*), 218
ChineseRemainderTheorem(I1, I2, e1, e2) (*function*), 122, 331
ChineseRemainderTheorem(I1, L1, e1, L2) (*function*), 122
ChineseRemainderTheorem(Is, as) (*function*), 218
ChineseRemainderTheorem(S, Z, V) (*function*), 398
ChineseRemainderTheorem(X, M) (*function*), 122
ChineseRemainderTheoremFunctions(Is) (*function*), 218
Class and Unit Groups of a Quartic Field (*example*), 100
Class Group and Unit Group (*example*), 26
Class Group of a Real Quadratic Field (*example*), 96
Class Groups by Sieving (*example*), 37
Class Groups of Relative Fields (*example*), 315
Classes(icm) (*function*), 225
Classes(W) (*function*), 223
ClassGroup(F : parameters) (*function*), 314, 355
ClassGroup(K) (*function*), 36
ClassGroup(K: parameters) (*function*), 93
ClassGroup(O) (*function*), 36, 314
ClassGroup(O: parameters) (*function*), 93

ClassGroupAbelianInvariants(F : parameters) (*function*), 314, 355
ClassGroupAbelianInvariants(O) (*function*), 314
ClassGroupCyclicFactorGenerators(O) (*function*), 96
ClassGroupExactSequence(F) (*function*), 356
ClassGroupExactSequence(O) (*function*), 314
ClassGroupGenerationBound(F) (*function*), 354
ClassGroupGenerationBound(q, g) (*function*), 354
ClassGroupGetUseMemory(O) (*function*), 98
ClassGroupPRank(F) (*function*), 315, 357
ClassGroupPrimeRepresentatives(O, I) (*function*), 95
ClassGroupSetUseMemory(O, f) (*procedure*), 98
ClassNumber(F) (*function*), 315, 355
ClassNumber(K) (*function*), 37
ClassNumber(K: parameters) (*function*), 95
ClassNumber(O) (*function*), 37, 315
ClassNumber(O: parameters) (*function*), 95
ClassNumberApproximation(F, e) (*function*), 354
ClassNumberApproximationBound(q, g, e) (*function*), 355
ClassRepresentative(I) (*function*), 122
CM Type of an Etale Algebra (*example*), 226
CMPosElt(PHI) (*function*), 226
CMPositiveElement(PHI) (*function*), 226
CMType(b) (*function*), 226
CMType(seq) (*function*), 226
Codifferent(I) (*function*), 117, 335
CoefficientField(A) (*function*), 188
CoefficientField(D) (*function*), 365
CoefficientField(F) (*function*), 10, 67, 288
CoefficientField(FF) (*function*), 288
CoefficientHeight(a) (*function*), 328
CoefficientHeight(E) (*function*), 88
CoefficientHeight(I) (*function*), 109, 335
CoefficientIdeals(I) (*function*), 111, 335
CoefficientIdeals(O) (*function*), 78, 292
CoefficientLength(a) (*function*), 328
CoefficientLength(E) (*function*), 88
CoefficientLength(I) (*function*), 109, 336
CoefficientRing(A) (*function*), 188
CoefficientRing(F) (*function*), 10, 67, 265, 288
CoefficientRing(FF) (*function*), 288
CoefficientRing(O) (*function*), 67, 288
CoefficientRing(R) (*function*), 153
CohenMacaulayType(S) (*function*), 221
CohenMacaulayTypeAtPrime(S, P) (*function*), 221
CohomologyModule(A) (*function*), 186
Colon Ideal of a Noninvertible Ideal (*example*), 117
ColonIdeal(I, J) (*function*), 117, 211, 331
ColonIdeal(I, O) (*function*), 211
ColonIdeal(O, J) (*function*), 211
CombineInvariants(R, G, H1, H2, H3) (*function*), 154
CommonZeros(F, L) (*function*), 326
CommonZeros(L) (*function*), 326
comp<K|P> (*constructor*), 13, 73
comp<O|P> (*constructor*), 73
ComplementaryDivisor(D) (*function*), 354
Completion(F, p) (*function*), 344
Completion(K, P) (*function*), 13, 73
Completion(O, P) (*function*), 73
Completion(O, p) (*function*), 344
Completion(P) (*function*), 216
Complex Embeddings of an Etale Algebra (*example*), 203
ComplexConjugate(a) (*function*), 40, 46
ComplexConjugate(I) (*function*), 226
ComplexConjugate(O) (*function*), 225
ComplexConjugate(x) (*function*), 18, 82, 225
ComplexImage(x) (*function*), 20
Component(chi, oo) (*function*), 230
Component(chi, P) (*function*), 230
Component(GR, oo) (*function*), 237
Component(GR, P) (*function*), 237
Component(psi, oo) (*function*), 230
Component(psi, P) (*function*), 230
Components and Idempotents of an Etale Algebra (*example*), 205
Components(A) (*function*), 187, 203
Components(chi) (*function*), 230
Components(F) (*function*), 10, 68
Components(GR) (*function*), 237
Components(O) (*function*), 68
Components(psi) (*function*), 230
Components(x) (*function*), 204
Composite(K, L) (*function*), 19
CompositeFields(F, L) (*function*), 6, 51
Composition(f, g) (*function*), 309
Compositum of Two Number Fields (*example*), 10, 68
Compositum(K, A) (*function*), 51
Compositum(K, L) (*function*), 6, 51
ConditionalClassGroup(K) (*function*), 94
ConditionalClassGroup(O) (*function*), 94
Conductor(A) (*function*), 188, 386, 404
Conductor(chi) (*function*), 230
Conductor(GR) (*function*), 237
Conductor(K) (*function*), 36, 45
Conductor(m) (*function*), 386
Conductor(m, U) (*function*), 386
Conductor(O) (*function*), 36, 77, 211
Conductor(psi) (*function*), 230
Conjugate Elements of an Algebraic Closure (*example*), 256
Conjugate(a) (*function*), 40
Conjugate(a, l) (*function*), 87
Conjugate(a, n) (*function*), 46
Conjugate(a, r) (*function*), 46
Conjugate(I) (*function*), 41

Conjugates(a) (*function*), 86, 255
ConjugatesToPowerSums(I) (*function*), 165
Constant and Coefficient Fields (*example*), 289
ConstantCoefficient(F) (*function*), 392
ConstantField(F) (*function*), 288
ConstantFieldExtension(F, E) (*function*), 290
Constructing Divisors in OM Representation (*example*), 374
Content(I) (*function*), 41, 110
Content(M) (*function*), 116
Converting OM Prime Ideals to Ideals (*example*), 368
Converting to and from OM Representation (*example*), 134
CoprimeBasis(L) (*function*), 121
CoprimeBasisInsert(~L, I) (*procedure*), 121
CoprimeRepresentative(I, J) (*function*), 122, 212
Counting Places of a Given Degree (*example*), 344
CreateCMType(b) (*function*), 226
CreateCMType(seq) (*function*), 226
Creating a Function Field Extension (*example*), 282
Creating a Non-Simple Extension (*example*), 282
Creating a Number Field (*example*), 4
Creating a Quadratic Field and Order (*example*), 35
Creating a Relative Extension (*example*), 282
Creating an Embedded Number Field (*example*), 20
Creating an OM Ideal (*example*), 372
Creating Class Fields of a Function Field (*example*), 384
Creating Function Field Elements (*example*), 320
Creating Number Field Elements (*example*), 8
Creating Orders of a Function Field (*example*), 284
Creating Places (*example*), 341
Creating Rational Function Field Elements (*example*), 264
Creating Roots in the Algebraic Closure (*example*), 248
CRT(I1, I2, e1, e2) (*function*), 122, 331
CRT(I1, L1, e1, L2) (*function*), 122
CRT(S, Z, V) (*function*), 398
CRT(X, M) (*function*), 122
CyclicToRadical(K, a, z) (*function*), 162, 304
Cyclotomic Field as a Class Field (*example*), 181
CyclotomicAutomorphismGroup(K) (*function*), 45
CyclotomicField(m) (*function*), 43
CyclotomicOrder(K) (*function*), 45
CyclotomicPolynomial(m) (*function*), 43
CyclotomicRelativeField(k, K) (*function*), 45

D

d * k (*operation*), 29, 130
D + P (*operation*), 346
d + p (*operation*), 29, 130
D - P (*operation*), 346
d - p (*operation*), 29, 130
D div k (*operation*), 346
d div k (*operation*), 29, 130
D in Div (*operation*), 347
D mod k (*operation*), 346
D notin Div (*operation*), 347
D1 + D2 (*operation*), 346, 375
d1 + d2 (*operation*), 29, 130
D1 - D2 (*operation*), 346, 375
d1 - d2 (*operation*), 29, 130
D1 eq D2 (*operation*), 347, 359, 377
d1 eq d2 (*operation*), 28, 129
D1 ge D2 (*operation*), 347
D1 gt D2 (*operation*), 347
D1 le D2 (*operation*), 347
D1 lt D2 (*operation*), 347
D1 ne D2 (*operation*), 347
Decomposing Artin Representations (*example*), 405
Decomposition of a Rational Function (*example*), 268
Decomposition(A) (*function*), 405
Decomposition(a) (*function*), 93
Decomposition(chi) (*function*), 230
Decomposition(f) (*function*), 268
Decomposition(F, P) (*function*), 340
Decomposition(K, I) (*function*), 28, 129
Decomposition(K, p) (*function*), 28, 129
Decomposition(m, p) (*function*), 28, 129
Decomposition(O) (*function*), 334
Decomposition(O, p) (*function*), 118, 334
DecompositionField(p) (*function*), 143
DecompositionField(p, A) (*function*), 189
DecompositionGroup(P) (*function*), 30, 131
DecompositionGroup(p) (*function*), 143
DecompositionGroup(p, A) (*function*), 189
DecompositionType(A, p) (*function*), 189, 387
DecompositionType(F, P) (*function*), 340
DecompositionType(m, U, p) (*function*), 387
DecompositionType(O) (*function*), 334
DecompositionType(O, p) (*function*), 118, 334
DecompositionTypeFrequency(A, a, b) (*function*), 189
DecompositionTypeFrequency(A, l) (*function*), 189
DefiningConstantField(F) (*function*), 288
DefiningPolynomial(A) (*function*), 203, 405
DefiningPolynomial(F) (*function*), 15, 76, 291
DefiningPolynomial(O) (*function*), 76, 291
DefiningPolynomials(F) (*function*), 291
DefiningPolynomials(O) (*function*), 291
Degree 4 Class Fields with Many Places (*example*), 385

Degree and Residue Field of OM Ideals (*example*), 373
Degree(A) (*function*), 188, 253, 387, 404
Degree(a) (*function*), 326
Degree(A, v) (*function*), 253
Degree(D) (*function*), 30, 32, 131, 349, 365, 377
Degree(e) (*function*), 166
Degree(F) (*function*), 14, 75, 291, 392
Degree(f) (*function*), 267
Degree(F, G) (*function*), 291
Degree(I) (*function*), 110, 337, 373
Degree(O) (*function*), 75, 291
Degree(P) (*function*), 30, 131, 343
DegreeOfExactConstantField(A) (*function*), 386
DegreeOfExactConstantField(F) (*function*), 292
DegreeOfExactConstantField(m) (*function*), 386
DegreeOfExactConstantField(m, U) (*function*), 386
DegreeOnePrimeIdeals(O, B) (*function*), 93
Degrees and Valuations of Approximants (*example*), 277
Denominator(a) (*function*), 22, 84, 325
Denominator(a, O) (*function*), 325
Denominator(D) (*function*), 349
Denominator(f) (*function*), 267
Denominator(f, R) (*function*), 267
Denominator(I) (*function*), 109, 335
Dense and Sparse Cyclotomic Fields (*example*), 44
Derivative(f) (*function*), 269
Derivative(f, k) (*function*), 269
Derivative(f, v) (*function*), 269
Derivative(f, v, k) (*function*), 269
Derivative(x, i) (*function*), 153
Determinant(A) (*function*), 403
Development(a, P) (*function*), 325
DiagonalEmbedding(K, V) (*function*), 204
Different(a) (*function*), 93, 328
Different(I) (*function*), 117, 335
Different(O) (*function*), 77, 295
DifferentDivisor(F) (*function*), 345
Differential Space of a Divisor (*example*), 359
Differential(a) (*function*), 358
DifferentialBasis(D) (*function*), 354, 359
Differentials as a Module (*example*), 361
DifferentialSpace(D) (*function*), 354, 359
DifferentialSpace(F) (*function*), 289, 358
Differentiation(x, a) (*function*), 328
Differentiation(x, n, a) (*function*), 328
DifferentiationSequence(x, n, a) (*function*), 328
Dimension(A) (*function*), 203, 404
Dimension(D) (*function*), 350, 377
DimensionOfExactConstantField(F) (*function*), 292
DirectProduct(seq) (*function*), 203
Dirichlet Character with Prescribed Values (*example*), 234
Dirichlet Characters over Q and Number Fields (*example*), 236
Dirichlet Characters over the Rationals (*example*), 232
DirichletCharacter(A) (*function*), 407
DirichletCharacter(G, B) (*function*), 233
DirichletCharacter(G, L) (*function*), 233
DirichletCharacter(I, B) (*function*), 233
DirichletCharacter(I, L) (*function*), 233
DirichletCharacter(I, oo, B) (*function*), 233
DirichletCharacter(I, oo, L) (*function*), 233
DirichletCharacterOverNF(chi) (*function*), 235
DirichletCharacterOverQ(chi) (*function*), 235
DirichletGroup(I) (*function*), 229
DirichletGroup(I, oo) (*function*), 229
DirichletRestriction(psi) (*function*), 232
Discriminant(A) (*function*), 188
Discriminant(F) (*function*), 15, 75
Discriminant(I) (*function*), 41
Discriminant(K) (*function*), 36
Discriminant(O) (*function*), 75, 292
DiscriminantDivisor(A) (*function*), 386
DiscriminantDivisor(m, U) (*function*), 386
DiscriminantRange(D) (*function*), 32
Div ! a (*operation*), 345
Div ! I (*operation*), 345
Div ! P (*operation*), 345
Div1 eq Div2 (*operation*), 346
Div1 ne Div2 (*operation*), 346
Divisor Class Group Computations (*example*), 355
Divisor(a) (*function*), 325, 345
Divisor(D) (*function*), 377
Divisor(d) (*function*), 360
Divisor(I) (*function*), 28, 129, 335, 345
Divisor(I, J) (*function*), 335, 345
Divisor(P) (*function*), 345
Divisor(pl) (*function*), 28, 129
Divisor(x) (*function*), 28, 130
DivisorGroup(F) (*function*), 289, 342, 345
DivisorGroup(K) (*function*), 28, 129
DivisorOfDegreeOne(F) (*function*), 346
Divisors on a Relative Function Field (*example*), 347
Divisors(a) (*function*), 93
Divisors(I) (*function*), 120
Domain(G) (*function*), 230
DotProduct(a, b) (*function*), 205
Drinfeld Module on an Elliptic Curve (*example*), 395

E

e * f (*operation*), 166
e eq f (*operation*), 166
E in I (*operation*), 116
e meet f (*operation*), 166

e subset f (*operation*), 166
ElementarySymmetricPolynomial(R, i) (*function*), 152
ElementToSequence(a) (*function*), 25, 92, 127, 320, 340
EllipticCurve(GR) (*function*), 245
elt< F | a > (*constructor*), 8, 61
elt< F | a, b > (*constructor*), 264
elt< F | a_0, a_1, ..., a_{m-1}> (*constructor*), 8, 62
elt< F | a_0, a_1, ..., a_{n-1}> (*constructor*), 319
elt< F | [a_0, a_1, ..., a_{m-1}] > (*constructor*), 8, 62
elt< FF | a_1, a_2, ..., a_n> (*constructor*), 319
elt< O | a > (*constructor*), 62
elt< O | a_0, a_1, ..., a_{m-1} > (*constructor*), 62
elt< O | a_1, a_2, ..., a_n> (*constructor*), 319
elt< O | [a_0, a_1, ..., a_{m-1}] > (*constructor*), 62
elt<F | a> (*constructor*), 318
elt<FF | a> (*constructor*), 319
elt<O | a> (*constructor*), 319
Eltseq(a) (*function*), 25, 92, 127, 320, 340, 388
Eltseq(a, R) (*function*), 320
Eltseq(E, k) (*function*), 25, 92
Eltseq(F) (*function*), 393
Embed(F, L, a) (*procedure*), 11, 71, 289
Embed(F, L, s) (*procedure*), 289
EmbeddedNumberField(f, i) (*function*), 19
EmbeddedNumberField(f, r) (*function*), 19
EmbeddedNumberField(L, c) (*function*), 19
EmbeddedNumberField(L, i) (*function*), 19
EmbeddedSplittingField(f) (*function*), 19
Embedding a Quadratic Field in a Cyclotomic Field (*example*), 35
Embedding a Quadratic Field in a Quartic Field (*example*), 9, 63
Embedding One Number Field in Another (*example*), 12, 72
Embedding(K) (*function*), 20
EmbeddingMap(e) (*function*), 166
EmbeddingMap(F, L) (*function*), 12, 71
EpsilonFactor(A) (*function*), 406
EpsilonFactor(A,infty) (*function*), 406
EpsilonFactor(A,p) (*function*), 406
Equality(f, g) (*function*), 309
Equation Order and Ring of Integers (*example*), 53
EquationOrder(A) (*function*), 186, 208
EquationOrder(F) (*function*), 35
EquationOrder(f) (*function*), 52
EquationOrder(K) (*function*), 53
EquationOrder(O) (*function*), 53, 283
EquationOrder(S) (*function*), 52
EquationOrderFinite(F) (*function*), 283
EquationOrderInfinite(F) (*function*), 283
Etale Algebra of Two Copies of Q (*example*), 202
EtaleAlgebra(f) (*function*), 202
EtaleAlgebra(seq) (*function*), 202
EuclideanNorm(a) (*function*), 127
EulerFactor(A, p) (*function*), 406
EulerProduct(O, B) (*function*), 96
Evaluate(a, P) (*function*), 325, 343
Evaluate(f) (*function*), 153
Evaluate(f, a) (*function*), 206
Evaluate(f, r) (*function*), 268
Evaluate(f, S) (*function*), 153, 268
Evaluate(f, S, m) (*function*), 153
Evaluate(f, v, r) (*function*), 268
Evaluate(t, a, b) (*function*), 105
Evaluate(t, S) (*function*), 105
Evaluate(x, p) (*function*), 29, 87
ExactConstantField(F) (*function*), 288
ExceptionalUnitOrbit(u) (*function*), 100
ExceptionalUnits(O) (*function*), 101
Exp(x, p) (*function*), 396
Expand(a, P) (*function*), 325
ext< F | s1, ..., sn > (*constructor*), 3, 50
ext< K | f > (*constructor*), 280
ext< O | a_1, ..., a_r > (*constructor*), 53
ext< O | f > (*constructor*), 53
ext< Q | s > (*constructor*), 3, 50
ext< Q | s1, ..., sn > (*constructor*), 3, 50
ext< Z | f > (*constructor*), 53
ext<F | s> (*constructor*), 3, 50
ext<O | f> (*constructor*), 284
Extend(chi, D) (*function*), 231
Extend(chi, I) (*function*), 231
Extend(chi, I, oo) (*function*), 231
Extend(D, x, p) (*function*), 394
Extend(G, D) (*function*), 231
Extend(G, H) (*function*), 231
Extend(G, I) (*function*), 231
Extend(G, I, oo) (*function*), 231
Extend(GR, I) (*function*), 238
Extend(psi, H) (*function*), 231
Extend(psi, I) (*function*), 231
Extend(psi, I, oo) (*function*), 231
ExtendedType(a) (*function*), 22
ExtendedType(K) (*function*), 9
Extending to the Exact Constant Field (*example*), 291
Extends(P, p) (*function*), 30, 131
ExtensionHomPicardGroups(S, T) (*function*), 219

F

F ! a (*operation*), 8, 61, 264, 318
F ! [a_0, a_1, ..., a_{n-1}] (*operation*), 320
F ! [a, b] (*operation*), 264
F ! [a_0, a_1, ..., a_{m-1}] (*operation*), 8, 62
F !! I (*operation*), 107
F . 1 (*operation*), 318
F . 2 (*operation*), 318
F . i (*operation*), 67
F eq G (*operation*), 266, 316

F eq L (*operation*), 17, 81
F ne G (*operation*), 266, 316
F ne L (*operation*), 18, 81
FactorBasis(K, B) (*function*), 96
FactorBasis(O) (*function*), 96
FactorBasis(O, B) (*function*), 96
FactorBasisVerify(O, a, b) (*procedure*), 95
Factoring an Ideal of a Maximal Order (*example*), 338
Factorisation(I) (*function*), 118, 137, 334, 372
Factorisation(n) (*function*), 40
Factorization in a Non-Maximal Order (*example*), 118
Factorization(I) (*function*), 118, 137, 220, 334, 372
Factorization(n) (*function*), 40
FF ! a (*operation*), 319
FF . i (*operation*), 318
Field of Definition of the Cyclic-6 Variety (*example*), 258
Field(A) (*function*), 404
FieldAutomorphism(F, a) (*function*), 169
FieldAutomorphism(F, g) (*function*), 169
FieldAutomorphism(K) (*function*), 169
FieldMorphism(f) (*function*), 309
FieldOfFractions(A) (*function*), 252
FieldOfFractions(F) (*function*), 288
FieldOfFractions(FF) (*function*), 288
FieldOfFractions(O) (*function*), 53, 288
FieldOfFractions(P) (*function*), 264
FindGenerators(G) (*function*), 200
FindOverOrders(R) (*function*), 215
FiniteDivisor(D) (*function*), 350
FiniteSplit(D) (*function*), 350
FixedField(a) (*function*), 169
FixedField(A, U) (*function*), 186
FixedField(K, S) (*function*), 143
FixedField(K, U) (*function*), 143
FixedFieldSymbol(a, P) (*function*), 169
FixedGroup(K, a) (*function*), 143
FixedGroup(K, L) (*function*), 143
Flat(a) (*function*), 320
Flat(e) (*function*), 26, 92
FractionalIdeal(I) (*function*), 108
FractionalIdeal(x) (*function*), 108
Frobenius Elements in a Galois Group (*example*), 147
FrobeniusAutomorphism(A, p) (*function*), 190
FrobeniusElement(K, p) (*function*), 147
FrobeniusImage(e) (*function*), 388
FrobeniusMap(W) (*function*), 388
FunctionField(A) (*function*), 384
FunctionField(D) (*function*), 349, 358
FunctionField(d) (*function*), 358
FunctionField(e) (*function*), 389
FunctionField(f : parameters) (*function*), 280, 281
FunctionField(G) (*function*), 346
FunctionField(O) (*function*), 288
FunctionField(P) (*function*), 343
FunctionField(R) (*function*), 263, 282
FunctionField(R, r) (*function*), 263
FunctionField(S) (*function*), 281, 342
FunctionFieldDatabase(q, d) (*function*), 365
FunctionFields(D) (*function*), 365
Functions on Differentials (*example*), 360
FundamentalUnit(K) (*function*), 36
FundamentalUnit(O) (*function*), 36
FundamentalUnits(O) (*function*), 315
Further Order Constructions (*example*), 286

G

Galois Action on S-Units (*example*), 124
Galois Group of a Sextic Polynomial (*example*), 151
Galois Group over a Rational Function Field (*example*), 296
Galois Groups in Positive Characteristic (*example*), 297
Galois Splitting Field of a Cubic (*example*), 157
Galois Splitting Field of a Polynomial (*example*), 300
GaloisAutomorphismGroup(K) (*function*), 150
GaloisGroup(F) (*function*), 296
GaloisGroup(f) (*function*), 148, 295
GaloisGroup(K) (*function*), 26, 148
GaloisGroupConjugation(S, T) (*function*), 150
GaloisGroupInvariant(G, H) (*function*), 153
GaloisProof(f, S) (*function*), 149
GaloisProof(K, S) (*function*), 149
GaloisQuotient(f, Q) (*function*), 155
GaloisQuotient(K, Q) (*function*), 155
GaloisQuotient(S, Q) (*function*), 155
GaloisRoot(f, i, S) (*function*), 149
GaloisRoot(i, S) (*function*), 149
GaloisSibling(S, phi) (*function*), 155
GaloisSplittingField(f) (*function*), 156, 300
GaloisSubfieldTower(S, L) (*function*), 156
GaloisSubgroup(f, U) (*function*), 155
GaloisSubgroup(K, U) (*function*), 155
GaloisSubgroup(S, U) (*function*), 155
GapNumbers(D) (*function*), 351
GapNumbers(D, P) (*function*), 351
GapNumbers(F) (*function*), 294
GapNumbers(F, P) (*function*), 294
GapNumbers(P) (*function*), 294
Gaussian Periods (*example*), 46
GCD(a, b) (*function*), 39
Gcd(a, b) (*function*), 39, 126
GCD(D1, D2) (*function*), 346, 375
Gcd(D1, D2) (*function*), 346, 375
GCD(F, G) (*function*), 392
GCD(I, J) (*function*), 116
Gcd(I, J) (*function*), 116, 333
gen * S (*operation*), 210

Generators(A) (*function*), 187
Generators(I) (*function*), 112, 210, 335
Generators(K) (*function*), 23, 86
Generators(K, k) (*function*), 24, 86
Generators(K, Q) (*function*), 24, 86
Generators(S) (*function*), 207
GeneratorsOverBaseRing(K) (*function*), 23
GeneratorsSequence(K) (*function*), 23
GeneratorsSequenceOverBaseRing(K) (*function*), 24
GenericGroup(X) (*function*), 200
Genus Distribution in the Function Field Database (*example*), 365
Genus(A) (*function*), 387
Genus(F) (*function*), 292
Genus(m, U) (*function*), 387
GenusField(A) (*function*), 185
Geometric Galois Group (*example*), 299
GeometricGaloisGroup(f) (*function*), 299
GetEvaluationComparison(R) (*function*), 153
GetUseMontes(t) (*function*), 133, 368
GHS Weil Descent (*example*), 363
GlobalUnitGroup(F) (*function*), 315, 356
GraphOverOrders(R) (*function*), 215
GreatestCommonDivisor(a, b) (*function*), 39
GreatestCommonDivisor(D1, D2) (*function*), 346, 375
GreatestCommonDivisor(I, J) (*function*), 116
GRHBound(K) (*function*), 95
GRHBound(O) (*function*), 95
Grossencharacter Central Value and Gamma Periods (*example*), 239
Grossencharacter from a Hypergeometric Motive (*example*), 242
Grossencharacter on the Eighth Cyclotomic Field (*example*), 242
Grossencharacter with Vanishing Central Value (*example*), 239
Grossencharacter(E) (*function*), 245
Grossencharacter(psi, chi, T) (*function*), 237
Grossencharacter(psi, T) (*function*), 237
Grossencharacters of CM Elliptic Curves (*example*), 245
Grossencharacters of Trivial Conductor (*example*), 241
Grossencharacters on the Gaussian Field (*example*), 238
GrossenTwist(GR, D) (*function*), 238
GroundField(F) (*function*), 10, 67
Group(A) (*function*), 404
Growth of Ray Class Groups (*example*), 176

H

H2_G_A(A) (*function*), 185
HasBaseField(A) (*function*), 203
HasComplexConjugate(A) (*function*), 225
HasComplexConjugate(K) (*function*), 18, 81
HasEmbeddedNumberField(f, r) (*function*), 19
HasEmbeddedNumberField(L, c) (*function*), 19
HasEmbedding(F, L) (*function*), 71
HasEmbedding(O1, O2) (*function*), 71, 289
HasInverse(f) (*function*), 309
HasPlace(F, m) (*function*), 312, 341
HasRandomPlace(F, m) (*function*), 313, 341
HasseWittInvariant(F) (*function*), 315, 357
HasTotallyPositiveGenerator(I) (*function*), 101
Hecke Characters of a Class Field (*example*), 187
HeckeCharacter(A) (*function*), 407
HeckeCharacter(G, B) (*function*), 233
HeckeCharacter(G, L) (*function*), 233
HeckeCharacter(I, B) (*function*), 233
HeckeCharacter(I, L) (*function*), 233
HeckeCharacter(I, oo, B) (*function*), 233
HeckeCharacter(I, oo, L) (*function*), 233
HeckeCharacterGroup(A) (*function*), 187
HeckeCharacterGroup(I) (*function*), 229
HeckeCharacterGroup(I, oo) (*function*), 229
HeckeCharacterGroup(L) (*function*), 187
HeckeLift(chi) (*function*), 232
Height(D) (*function*), 377
Hermitian Function Field (*example*), 283
HermitianFunctionField(p, d) (*function*), 281
HermitianFunctionField(q) (*function*), 281
Hilbert Class Field of a Cubic Field (*example*), 173
Hilbert Class Field of a Sextic Field (*example*), 183
Hilbert Class Field of an Imaginary Quadratic Field (*example*), 195
Hilbert90(a, M) (*function*), 167
HilbertCharacterSubgroup(G) (*function*), 229
HilbertClassField(K) (*function*), 183
HilbertClassField(K, p) (*function*), 384
HilbertIrreducibilityCurves(f) (*function*), 299
Hom(A, B, img) (*function*), 203
hom< F -> R | h, r > (*constructor*), 9, 63
hom< F -> R | r > (*constructor*), 9, 63
hom< O -> R | b_1, ..., b_n > (*constructor*), 64, 318
hom< O -> R | cf, g > (*constructor*), 317
hom< O -> R | g > (*constructor*), 317
hom< O -> R | h, r > (*constructor*), 63
hom< O -> R | m, b_1, ..., b_n > (*constructor*), 64, 318
hom< O -> R | r > (*constructor*), 63
hom< P -> S | f, y_1, ..., y_n > (*constructor*), 266
hom< P -> S | y_1, ..., y_n > (*constructor*), 266
hom<F -> R | cf, g> (*constructor*), 317
hom<F -> R | g> (*constructor*), 317
Homomorphism into a Product Algebra (*example*), 204
Homomorphism to a Number Field (*example*), 266

Homomorphisms of Relative Function Fields (*example*), 317
Homs(PHI) (*function*), 226
HomsToC(A) (*function*), 203

I

I * c (*operation*), 331
I * J (*operation*), 115, 134, 211, 330, 369
I * x (*operation*), 115, 211
I + J (*operation*), 116, 134, 211, 330, 369
I / c (*operation*), 331
I / J (*operation*), 116, 134, 330, 369
I / x (*operation*), 116
I @ a (*operation*), 170
I ^ k (*operation*), 116, 330
I ^ n (*operation*), 134, 211, 369
I div J (*operation*), 115
I eq J (*operation*), 113, 116, 135, 210, 332, 370
I eq S (*operation*), 210
I in S (*operation*), 332
I meet J (*operation*), 116, 211, 333
I meet R (*operation*), 117, 335
I meet S (*operation*), 211
I ne J (*operation*), 113, 210, 332
I notin S (*operation*), 332
I subset J (*operation*), 113, 116, 135, 370
I subset S (*operation*), 211
I1 subset I2 (*operation*), 211
icm ! x (*operation*), 225
ICM(S) (*function*), 224
icm1 eq icm2 (*operation*), 225
ICM_bar(S) (*function*), 224
Id(D) (*function*), 358
Id(G) (*function*), 345
Ideal Generated by an Element (*example*), 108
Ideal Operations in a Relative Field (*example*), 336
Ideal(A) (*function*), 253
Ideal(D) (*function*), 29, 130
Ideal(f) (*function*), 41
Ideal(I) (*function*), 133, 368
Ideal(P) (*function*), 330, 344
Ideal(S, gen) (*function*), 210
Ideal(S, gens) (*function*), 210
Ideal(S, idls) (*function*), 210
Ideal(x) (*function*), 223, 225
ideal< O | a_1, a_2, ... , a_m > (*constructor*), 108, 330
ideal< O | T, d > (*constructor*), 330
ideal< O | T, I_1, ..., I_n > (*constructor*), 330
ideal< O | T, S > (*constructor*), 330
ideal<O | M, d> (*constructor*), 108
ideal<O | M, I1, ..., In> (*constructor*), 108
ideal<O | M, [I1, ..., In]> (*constructor*), 108
ideal<O | x> (*constructor*), 108
IdealClassMonoidAbstract(R) (*function*), 225
IdealQuotient(I, J) (*function*), 117, 331
Ideals(D) (*function*), 330, 350
IdealsOfIndex(I, N) (*function*), 217
IdealsOfIndex(O, N) (*function*), 217
IdealsUpTo(B, K) (*function*), 122
IdealsUpTo(B, O) (*function*), 122
Idempotents(A) (*function*), 207
Idempotents(I, J) (*function*), 122
Idempotents(W) (*function*), 223
Identity(A) (*function*), 247
Identity(D) (*function*), 358
Identity(F) (*function*), 264, 319
Identity(G) (*function*), 345
Identity(K) (*function*), 9, 62
Identity(O) (*function*), 62, 319
IdentityAutomorphism(F) (*function*), 169
IdentityFieldMorphism(F) (*function*), 309
IharaBound(F) (*function*), 312
IharaBound(F, m) (*function*), 312
IharaBound(q, g) (*function*), 312
Implementing the Round 2 Algorithm (*example*), 60
ImproveAutomorphismGroup(F, E) (*function*), 190
IndependentUnits(K) (*function*), 99
IndependentUnits(O) (*function*), 99, 315
Index(a) (*function*), 93
Index(J, I) (*function*), 211
Index(O, I) (*function*), 109
Index(O, S) (*function*), 76, 295
Index(S, I) (*function*), 211
Index(S, T) (*function*), 209
Index(T) (*function*), 209, 211
IndexFormEquation(O, k) (*function*), 106
IndexOfSpeciality(D) (*function*), 350
Induced Map on Ray Class Groups (*example*), 179
InducedAutomorphism(r, h, c) (*function*), 179
InducedMap(m1, m2, h, c) (*function*), 179
InertiaDegree(I) (*function*), 110, 337
InertiaDegree(P) (*function*), 30, 131, 220, 343
InertiaField(p) (*function*), 144
InertiaGroup(p) (*function*), 143
InfiniteDivisor(D) (*function*), 350
InfinitePlaces(F) (*function*), 341
InfinitePlaces(K) (*function*), 28, 87, 129
InfinitePlaces(O) (*function*), 87, 129
InitializeEvaluation(R, S) (*procedure*), 153
IntegerRing(F) (*function*), 8, 35, 57, 265
IntegerRing(O) (*function*), 53
Integers(F) (*function*), 8, 57
Integers(O) (*function*), 53
Integral Basis of an OM Ideal (*example*), 371
Integral Closure in a Function Field (*example*), 285
Integral Model of a Number Field (*example*), 7
Integral Power Bases via Index Forms (*example*), 107
IntegralBasis(F) (*function*), 16, 77
IntegralBasis(F, R) (*function*), 16, 77

IntegralClosure(R, F) (*function*), 283
IntegralModel(F) (*function*), 7
IntegralNormEquation(a, N, O) (*function*), 103
IntegralSplit(a, O) (*function*), 324
IntegralSplit(I) (*function*), 117, 335
IntermediateIdeals(I, J) (*function*), 216
IntermediateIdeals(I, J, N) (*function*), 217
IntermediateIdeals(I, J, O) (*function*), 216
Intersection(K, L) (*function*), 20
Invariants of a Global Function Field (*example*), 313
Invariants of a Global Relative Field (*example*), 313
Invariants of a Non-Simple Field (*example*), 293
Invariants of a Relative Function Field (*example*), 292
Invariants of an Ideal (*example*), 110
Invariants of Artin Representations (*example*), 407
Inverse(a) (*function*), 169
Inverse(I) (*function*), 212
Inverse(x) (*function*), 205
InverseMod(E, M) (*function*), 117
ISABaseField(F,G) (*function*), 288
IsAbelian(A) (*function*), 185
IsAbelian(F) (*function*), 18, 82
IsAbsoluteField(K) (*function*), 18, 82
IsAbsoluteOrder(O) (*function*), 82, 317
IsAlgebraicField(R) (*function*), 17, 81
IsBass(S) (*function*), 220
IsBassAtPrime(S, P) (*function*), 220
IsCanonical(D) (*function*), 347, 358
IsCentral(A) (*function*), 185
IsCoercible(A, x) (*function*), 204
IsCoercible(S, x) (*function*), 207
IsCommutative(A) (*function*), 253
IsCommutative(F) (*function*), 265
IsCommutative(R) (*function*), 17, 81
IsComplex(p) (*function*), 30, 131
IsConjugateStable(I) (*function*), 226
IsConjugateStable(O) (*function*), 225
IsConstant(a) (*function*), 323
IsCoprime(I, J) (*function*), 212
IsCyclic(F) (*function*), 18, 82
IsDivisibleBy(a, b) (*function*), 322
IsDivisionRing(A) (*function*), 253
IsDivisionRing(F) (*function*), 266
IsDivisionRing(O) (*function*), 316
IsDivisionRing(R) (*function*), 316
IsDomain(A) (*function*), 254
IsDomain(F) (*function*), 266
IsDomain(O) (*function*), 316
IsDomain(R) (*function*), 18, 81, 316
IsEasySnAn(f) (*function*), 148
IsEffective(D) (*function*), 347, 377
IsEquationOrder(O) (*function*), 82, 317
IsEuclideanDomain(A) (*function*), 253
IsEuclideanDomain(F) (*function*), 17, 81, 265
IsEuclideanDomain(R) (*function*), 316
IsEuclideanRing(A) (*function*), 254
IsEuclideanRing(F) (*function*), 266
IsEuclideanRing(R) (*function*), 316
IsEven(chi) (*function*), 232
IsExact(d) (*function*), 360
IsExceptionalUnit(u) (*function*), 100
IsField(A) (*function*), 253
IsField(F) (*function*), 265
IsField(O) (*function*), 316
IsField(R) (*function*), 17, 81, 316
IsFinite(A) (*function*), 253
IsFinite(F) (*function*), 265
IsFinite(P) (*function*), 343
IsFinite(p) (*function*), 30, 130
IsFinite(R) (*function*), 17, 81
IsFiniteOrder(O) (*function*), 316
IsGeneratingSet(W, seq) (*function*), 223
IsGlobal(F) (*function*), 316
IsGloballySplit(C, l) (*function*), 168
IsGlobalUnit(a) (*function*), 323, 356
IsGlobalUnitWithPreimage(a) (*function*), 323, 356
IsGorenstein(O) (*function*), 220
IsGorensteinAtPrime(S, P) (*function*), 220
IsIdempotent(a) (*function*), 23, 86, 255, 267, 322
IsIdempotent(x) (*function*), 223
IsIdentity(a) (*function*), 169
IsIdentity(f) (*function*), 309
IsInArtinSchreierRepresentation(K) (*function*), 317
IsInert(P) (*function*), 115, 332
IsInert(P, O) (*function*), 115, 332
IsInfinite(p) (*function*), 30, 130
IsInKummerRepresentation(K) (*function*), 317
IsInt(x, B, S) (*function*), 150
IsIntegral(a) (*function*), 23, 85
IsIntegral(I) (*function*), 113, 135, 212, 332, 370
IsIntegral(x) (*function*), 206
IsInvariant(F, p) (*function*), 154
IsInvertible(I) (*function*), 212
IsInvertibleInMultiplicatorRing(x) (*function*), 225
IsIrreducible(A) (*function*), 406
IsIrreducible(a) (*function*), 23, 86, 255, 322
IsIsomorphic(F, L) (*function*), 18, 82
IsIsomorphic(I, J) (*function*), 220
IsIsomorphic(K, E) (*function*), 308
IsIsomorphicOverQt(K, L) (*function*), 307
IsKnownOrder(~R) (*procedure*), 208
IsLocalNorm(A, x) (*function*), 192
IsLocalNorm(A, x, i) (*function*), 192
IsLocalNorm(A, x, p) (*function*), 192
IsLocalSquare(a, p) (*function*), 91
IsMaximal(I) (*function*), 220
IsMaximal(O) (*function*), 82, 317
IsMaximal(S) (*function*), 208

IsMaximalAtPrime(R, P) (*function*), 214
IsMaximalIdeal(I) (*function*), 220
IsMinusOne(a) (*function*), 23, 85, 127, 255, 267, 322, 340
IsMinusOne(w) (*function*), 85
IsMorphism(f) (*function*), 309
IsNilpotent(a) (*function*), 23, 85, 255, 267, 322
IsNorm(A, x) (*function*), 192
IsNormal(A) (*function*), 185
IsNormal(F) (*function*), 18, 82
IsNumberField(A) (*function*), 203
IsNumberField(R) (*function*), 17, 81
IsOdd(chi) (*function*), 231
Isomorphisms of Function Fields (*example*), 309
Isomorphisms(K, E) (*function*), 307
Isomorphisms(K,E,p1,p2) (*function*), 308
IsOne(a) (*function*), 23, 85, 127, 254, 267, 322, 340
IsOne(I) (*function*), 113, 135, 332, 370
IsOne(w) (*function*), 85
IsOne(x) (*function*), 223, 225
IsOrdered(A) (*function*), 253
IsOrdered(F) (*function*), 265
IsOrdered(R) (*function*), 17, 81
IsPerfect(F) (*function*), 291
IsPID(A) (*function*), 253
IsPID(F) (*function*), 18, 81, 265
IsPID(O) (*function*), 81
IsPID(R) (*function*), 316
IsPositive(D) (*function*), 347, 377
IsPower(a, k) (*function*), 22, 84
IsPower(a, n) (*function*), 248
IsPower(I, k) (*function*), 118
IsPower(I, n) (*function*), 331
IsPower(w, n) (*function*), 85
IsPrime(a) (*function*), 23, 86, 255, 322
IsPrime(I) (*function*), 113, 135, 220, 332, 370
IsPrimitive(a) (*function*), 23, 85
IsPrimitive(chi) (*function*), 232
IsPrimitive(GR) (*function*), 237
IsPrimitive(psi) (*function*), 232
IsPrincipal(D) (*function*), 347, 377
IsPrincipal(I) (*function*), 113, 332
IsPrincipal(I1) (*function*), 219
IsPrincipalIdealRing(A) (*function*), 254
IsPrincipalIdealRing(F) (*function*), 18, 81, 266
IsPrincipalIdealRing(O) (*function*), 81
IsPrincipalIdealRing(R) (*function*), 316
IsProductOfIdeals(I) (*function*), 212
IsProductOfOrders(O) (*function*), 208
IsProductOfOrdersInComponents(O) (*function*), 208
IsProductOfOrdersInFactorAlgebras(S) (*function*), 208
IsQuadratic(K) (*function*), 35, 83
IsQuadratic(O) (*function*), 35
IsRamified(A, p) (*function*), 406
IsRamified(P) (*function*), 114, 332
IsRamified(P, O) (*function*), 114, 333
IsRationalFunctionField(F) (*function*), 316
IsReal(a) (*function*), 46
IsReal(p) (*function*), 30, 131
IsRegular(a) (*function*), 23, 86, 255, 268, 322
IsRingHomomorphism(m) (*function*), 64, 317
IsSeparating(a) (*function*), 322
IsSimple(F) (*function*), 18, 81
IsSimple(O) (*function*), 81
IsSpecial(D) (*function*), 347
IsSplit(P) (*function*), 115, 333
IsSplit(P, O) (*function*), 115, 333
IsSplitAsIdealAt(I, l) (*function*), 168
IsSPrincipal(D, S) (*function*), 357
IsSquare(a) (*function*), 22, 84, 248
IsSquare(I) (*function*), 118, 331
IsSquarefree(I) (*function*), 114
IsSubfield(F, L) (*function*), 18, 82
IsSubfield(K, L) (*function*), 307
IsSubfieldEmb(K, L) (*function*), 20
IsSUnit(a, S) (*function*), 356
IsSUnitWithPreimage(a, S) (*function*), 357
IsTamelyRamified(K) (*function*), 83
IsTamelyRamified(O) (*function*), 82, 317
IsTamelyRamified(P) (*function*), 114, 333
IsTamelyRamified(P, O) (*function*), 114, 333
IsTorsionUnit(w) (*function*), 85
IsTotallyEven(chi) (*function*), 232
IsTotallyPositive(a) (*function*), 23, 85
IsTotallyRamified(K) (*function*), 114
IsTotallyRamified(O) (*function*), 114, 317
IsTotallyRamified(P) (*function*), 114, 333
IsTotallyRamified(P, O) (*function*), 114, 333
IsTotallyReal(a) (*function*), 227
IsTotallyReal(K) (*function*), 83
IsTotallyRealPositive(a) (*function*), 227
IsTotallySplit(P) (*function*), 115, 333
IsTotallySplit(P, O) (*function*), 115, 333
IsTrivial(chi) (*function*), 231
IsTrivial(psi) (*function*), 231
IsTrivialOnUnits(chi) (*function*), 231
IsUFD(A) (*function*), 253
IsUFD(F) (*function*), 18, 81, 266
IsUFD(O) (*function*), 81
IsUFD(R) (*function*), 316
IsUnit(a) (*function*), 23, 85, 127, 255, 268, 322, 340
IsUnit(w) (*function*), 85
IsUnit(x) (*function*), 205
IsUnitary(A) (*function*), 253
IsUnitary(F) (*function*), 265
IsUnitary(R) (*function*), 17, 81
IsUnitWithPreimage(a) (*function*), 323
IsUnramified(K) (*function*), 83
IsUnramified(O) (*function*), 82, 317
IsUnramified(P) (*function*), 114, 333
IsUnramified(P, O) (*function*), 114, 333

IsWeakEquivalent(I, J) (*function*), 223
IsWeakEquivalent(J, O) (*function*), 224
IsWeakEquivalent(O, J) (*function*), 224
IsWeakEquivalent(O1, O2) (*function*), 223
IsWeaklyEquivalent(I, J) (*function*), 223
IsWeaklyEquivalent(J, O) (*function*), 224
IsWeaklyEquivalent(O1, O2) (*function*), 223
IsWeierstrassPlace(D, P) (*function*), 353
IsWeierstrassPlace(F, P) (*function*), 343
IsWeierstrassPlace(P) (*function*), 343
IsWildlyRamified(A, p) (*function*), 406
IsWildlyRamified(K) (*function*), 83
IsWildlyRamified(O) (*function*), 82, 317
IsWildlyRamified(P) (*function*), 114, 333
IsWildlyRamified(P, O) (*function*), 114, 333
IsZero(A) (*function*), 392
IsZero(a) (*function*), 23, 85, 126, 254, 267, 322, 340
IsZero(D) (*function*), 347
IsZero(d) (*function*), 360
IsZero(I) (*function*), 113, 135, 332, 370
IsZero(w) (*function*), 85
IsZeroDivisor(a) (*function*), 23, 86, 255, 268, 322
IsZeroDivisor(x) (*function*), 205

K

K !! A (*operation*), 403
K !! ch (*operation*), 403
k * D (*operation*), 346, 375
k * P (*operation*), 343
K . 1 (*operation*), 23, 86
K . i (*operation*), 10, 67, 264
K meet L (*operation*), 20
K subset L (*operation*), 18, 81
Kernel(A) (*function*), 405
Knot(A) (*function*), 192
KSpace(K, J) (*function*), 13, 73
KSpace(K, J, S) (*function*), 13, 73

L

L ! n (*operation*), 166
L-Function of a Hecke Character (*example*), 236
Lattice(I) (*function*), 88
Lattice(O) (*function*), 87
LCM(a, b) (*function*), 40
Lcm(a, b) (*function*), 40, 126
LCM(D1, D2) (*function*), 346
Lcm(D1, D2) (*function*), 346
LCM(I, J) (*function*), 116
Lcm(I, J) (*function*), 116, 334
LeadingCoefficient(F) (*function*), 392
LeastCommonMultiple(a, b) (*function*), 40
LeastCommonMultiple(D1, D2) (*function*), 346
LeastCommonMultiple(I, J) (*function*), 116
Length(a) (*function*), 88
Length(W) (*function*), 388
Lift(a, P) (*function*), 325, 344
Lift(i, P) (*function*), 344
Linear Relations among Roots (*example*), 163
LinearRelations(f) (*function*), 162
LinearRelations(f, I) (*function*), 163
LLL Reduction of an Order Basis (*example*), 69
LLL(O) (*function*), 69
LMFDBIdeal(K, s) (*function*), 113
LMFDBLabel(I) (*function*), 113
LoadWKICM(str) (*function*), 228
Local Root Numbers for Curve 27a (*example*), 243
LocalDegree(P) (*function*), 30, 131
Localization(O, P) (*function*), 73
Localization(O, p) (*function*), 291
Localization(W, P) (*function*), 223
LocalMultiplicativeGroupModSquares(p) (*function*), 91
LocalRing(P, prec) (*function*), 73
LocalRing(W) (*function*), 389
LocalUniformizer(P) (*function*), 344
Logs(a) (*function*), 87
LPolynomial(F) (*function*), 312
LPolynomial(F, m) (*function*), 312
L[n] (*literal*), 166

M

M @ a (*operation*), 170
MakeCoprime(I, J) (*function*), 122
MakeIntegral(I) (*function*), 212
Maximal Order with Known Ramification (*example*), 57
Maximal Order with new driver (*example*), 59
MaximalAbelianSubfield(F) (*function*), 183
MaximalAbelianSubfield(K) (*function*), 183, 384
MaximalAbelianSubfield(M) (*function*), 183
MaximalOrder(A) (*function*), 186, 208
MaximalOrder(F) (*function*), 8, 35, 57
MaximalOrder(f) (*function*), 57
MaximalOrder(O) (*function*), 57, 284
MaximalOrderFinite(A) (*function*), 384
MaximalOrderFinite(F) (*function*), 283
MaximalOrderFinite(u) (*function*), 389
MaximalOrderInfinite(A) (*function*), 384
MaximalOrderInfinite(F) (*function*), 283
MaximalOrderInfinite(u) (*function*), 389
MaximalOrders(u) (*function*), 389
MaximalSubfields(e) (*function*), 166
Maximum Degree of a Sequence (*example*), 272
MaximumDegree(f) (*function*), 272
MergeFields(F, L) (*function*), 6, 51
MergeUnits(K, a) (*function*), 100
MergeUnits(O, a) (*function*), 100
Min(a, O) (*function*), 325
Min(I) (*function*), 109
Minimal Polynomial and Representation Matrix (*example*), 326

Minimal Polynomial over a Subfield (*example*), 13, 74
Minimal Vector Sequence of Higher Order (*example*), 274
Minimal Vector Sequence of Polynomials (*example*), 273
Minimal Vector Sequence of Power Series (*example*), 273
Minimal Vector Sequence with Distortion (*example*), 274
MinimalCyclotomicField(a) (*function*), 44
MinimalCyclotomicField(S) (*function*), 44
MinimalField(a) (*function*), 44
MinimalField(S) (*function*), 44
MinimalInteger(I) (*function*), 109, 212
MinimalOverfields(e) (*function*), 167
MinimalOverOrders(R) (*function*), 214
MinimalOverOrdersAtPrime(R, P) (*function*), 214
MinimalPolynomial(a) (*function*), 24, 90, 255, 323
MinimalPolynomial(a, R) (*function*), 24, 90, 324
MinimalPolynomial(x) (*function*), 206
MinimalPolynomial(x, F) (*function*), 206
MinimalVectorSequence(f,n) (*function*), 273
Minimise(~a) (*procedure*), 44
Minimise(~s) (*procedure*), 45
Minimise(a) (*function*), 45
Minimise(s) (*function*), 45
Minimising an Artin Representation (*example*), 406
Minimize(~a) (*procedure*), 44
Minimize(~s) (*procedure*), 45
Minimize(A) (*function*), 405
Minimize(a) (*function*), 45
Minimize(s) (*function*), 45
Minimum(a, O) (*function*), 325
Minimum(I) (*function*), 109, 335
Minimum(P) (*function*), 343
MinkowskiBound(K) (*function*), 95
MinkowskiBound(O) (*function*), 95
MinkowskiLattice(I) (*function*), 88
MinkowskiLattice(O) (*function*), 87
MinkowskiSpace(F) (*function*), 12, 88
Modexp(a, e, n) (*function*), 40
Modexp(a, k, m) (*function*), 321
Modexp(a, n, m) (*function*), 84
Modinv(a, m) (*function*), 321
Modinv(E, M) (*function*), 117
Module and Relations over Relative Fields (*example*), 327
Module(I) (*function*), 112
Module(L, R) (*function*), 327, 360
Modulus(chi) (*function*), 230
Modulus(G) (*function*), 230
Modulus(GR) (*function*), 237
Modulus(OQ) (*function*), 126, 339
MonicModel(F) (*function*), 291
Montes Algorithm for a Function Field (*example*), 366
Montes Algorithm for a Polynomial (*example*), 366
Montes Algorithm on a Number Field (*example*), 132
Montes Algorithm on a Polynomial (*example*), 132
Montes(f, p) (*function*), 132, 366
Montes(K, p) (*procedure*), 132, 366
Multiplication Table of an Order (*example*), 80
MultiplicationTable(O) (*function*), 80
MultiplicationTable(W) (*function*), 223
MultiplicativeGroup(K) (*function*), 99
MultiplicativeGroup(O) (*function*), 99
MultiplicativeGroup(OQ) (*function*), 126
MultiplicatorRing(I) (*function*), 60, 212, 287, 334
MultiplicatorRing(R) (*function*), 209
MultiplicatorRing(x) (*function*), 223, 225
MultiplyFrobenius(b,f,F) (*function*), 363

N

N eq O (*operation*), 81
Name(F, 1) (*function*), 36
Name(F, i) (*function*), 67, 264, 318
Name(K, i) (*function*), 10, 67
Name(O, 1) (*function*), 36
Name(R, i) (*function*), 152
NaturalAction(K, V) (*function*), 204
NiceUnitSquareClassRepresentative(u, p) (*function*), 92
NonGorensteinPrimes(S) (*function*), 221
NonInvertiblePrimes(R) (*function*), 220
NonSpecialDivisor(m) (*function*), 399
Norm and Trace of Elements (*example*), 324
Norm Equation in a Quadratic Order (*example*), 39
Norm Equations via Class Fields (*example*), 192
Norm(a) (*function*), 24, 89, 255, 323
Norm(a, R) (*function*), 24, 89, 323
Norm(D) (*function*), 350
Norm(I) (*function*), 109, 137, 335, 371
Norm(P) (*function*), 344
Norm(P, a) (*function*), 170
Norm(x) (*function*), 215
Norm, Trace, and Minimal Polynomial (*example*), 25, 90
NormAbs(a) (*function*), 24, 89
NormAbs(I) (*function*), 109
Normal Closure via Composite Fields (*example*), 6
NormalClosure(F) (*function*), 5, 50
NormalSubfields(A) (*function*), 186
NormEquation(A, x) (*function*), 192
NormEquation(F, m) (*function*), 38, 102

NormEquation(F, m: parameters) (*function*), 38
NormEquation(m, N) (*function*), 103
NormEquation(O, m) (*function*), 38, 101
NormEquation(O, m: parameters) (*function*), 38
NormGroup(A) (*function*), 188
NormGroup(F) (*function*), 399
NormInduction(K, chi) (*function*), 232
NumberField(A) (*function*), 186, 257
NumberField(a) (*function*), 257
NumberField(D) (*function*), 28, 30, 129, 131
NumberField(e) (*function*), 166
NumberField(F) (*function*), 54
NumberField(f) (*function*), 2, 49
NumberField(O) (*function*), 35, 54
NumberField(P) (*function*), 28, 30, 129, 131
NumberField(s) (*function*), 3, 49
NumberFieldDatabase(d) (*function*), 31
NumberFields(D) (*function*), 32
NumberFields(D, d) (*function*), 32
NumberOfFields(D) (*function*), 32, 365
NumberOfFields(D, d) (*function*), 32
NumberOfPlacesDegECF(F, m) (*function*), 311, 342
NumberOfPlacesOfDegreeOne(A) (*function*), 387
NumberOfPlacesOfDegreeOne(m, U) (*function*), 387
NumberOfPlacesOfDegreeOneECF(F) (*function*), 311
NumberOfPlacesOfDegreeOneECF(F, m) (*function*), 312, 342
NumberOfPlacesOfDegreeOneECFBound(F) (*function*), 312
NumberOfPlacesOfDegreeOneECFBound(F, m) (*function*), 312, 342
NumberOfPlacesOfDegreeOneECFBound(q, g) (*function*), 312
NumberOfPlacesOfDegreeOneOverExactConstantField(F) (*function*), 311
NumberOfPlacesOfDegreeOneOverExactConstantField(F, m) (*function*), 312, 342
NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(F) (*function*), 312
NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(F, m) (*function*), 312, 342
NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(q, g) (*function*), 312
NumberOfPlacesOfDegreeOverExactConstantField(F, m) (*function*), 311, 342
NumberOfSmoothDivisors(n, m, P) (*function*), 346
Numerator(a) (*function*), 22, 84, 324
Numerator(a, O) (*function*), 324
Numerator(D) (*function*), 349
Numerator(f) (*function*), 267
Numerator(f, R) (*function*), 267

O

O ! a (*operation*), 62, 319
O ! [a_1, a_2, ..., a_n] (*operation*), 320
O ! [a_0, a_1, ..., a_{m-1}] (*operation*), 62
O !! I (*operation*), 107, 330
O * x (*operation*), 107, 330
O + P (*operation*), 55
O . i (*operation*), 67, 318
O meet P (*operation*), 55
O ne N (*operation*), 81
O subset P (*operation*), 81
O1 * O2 (*operation*), 209
O1 + O2 (*operation*), 286
O1 eq O2 (*operation*), 207, 316
O1 meet O2 (*operation*), 209, 286
O1 ne O2 (*operation*), 316
O1 subset O2 (*operation*), 209, 316
o`CyclotomicExtensions (*attribute*), 194
OMDivisor(D) (*function*), 374
OMDivisor(I) (*function*), 374
OMDivisor(If, Ii) (*function*), 374
OMDivisor(z) (*function*), 374
OMDivisorOfDegreeOne(F) (*function*), 374
OMRepresentation(I) (*function*), 134, 368
OMRepresentation(L, a) (*function*), 134, 368
OMRepresentation(L, S) (*function*), 134, 368
One(A) (*function*), 204, 247
One(F) (*function*), 264, 319
One(icm) (*function*), 225
One(K) (*function*), 9, 62
One(O) (*function*), 62, 319
One(R) (*function*), 390
One(S) (*function*), 208
One(W) (*function*), 223, 388
One-Dimensional Artin Representations (*example*), 407
OneIdeal(S) (*function*), 211
Operations on OM Divisors (*example*), 378
Operations on OM Ideals (*example*), 137
OptimisedRepresentation(F) (*function*), 7, 51
OptimisedRepresentation(F, d) (*function*), 51
OptimisedRepresentation(O) (*function*), 55
OptimisedRepresentation(O, d) (*function*), 55
Optimized Representation of a Field (*example*), 7, 51
OptimizedRepresentation(A) (*function*), 405
OptimizedRepresentation(F) (*function*), 7, 51
OptimizedRepresentation(F, d) (*function*), 51
OptimizedRepresentation(O) (*function*), 55
OptimizedRepresentation(O, d) (*function*), 55
OQ ! a (*operation*), 126, 339
Order of Maximal Cohen-Macaulay Type (*example*), 221
Order([e_1, ... e_n]) (*function*), 56
Order(a) (*function*), 169
Order(A, orders) (*function*), 207
Order(chi) (*function*), 230
Order(F) (*function*), 53
Order(FF) (*function*), 288
Order(gens) (*function*), 207

Order(I) (*function*), 109, 210, 334
Order(icm) (*function*), 225
Order(O, M) (*function*), 55, 285
Order(O, S) (*function*), 285
Order(O, T, d) (*function*), 55, 285
Order(psi) (*function*), 230
Order(W) (*function*), 223
Orders in a Product of Number Fields (*example*), 209
OrthogonalIdempotents(A) (*function*), 207
Over-Orders of an Equation Order (*example*), 214
OverOrders(R) (*function*), 214
OverOrdersAtPrime(R, P) (*function*), 214

P

p * k (*operation*), 29, 130
P + D (*operation*), 346
p + d (*operation*), 29, 130
P - D (*operation*), 346
p - d (*operation*), 29, 130
P div k (*operation*), 343
p div k (*operation*), 29, 130
P in S (*operation*), 343
P mod k (*operation*), 343
P notin S (*operation*), 343
P1 + P2 (*operation*), 343
p1 + p2 (*operation*), 29, 130
P1 - P2 (*operation*), 343
p1 - p2 (*operation*), 29, 130
P1 eq P2 (*operation*), 343
p1 eq p2 (*operation*), 28, 129
P1 ne P2 (*operation*), 343
p-Integral Basis of an OM Ideal (*example*), 136
Pade-Hermite Approximant with Distortion (*example*), 276
Pade-Hermite Approximants (*example*), 275
Pade-Hermite Approximants of Vectors (*example*), 278
PadeHermiteApproximant(f,d) (*function*), 275
PadeHermiteApproximant(f,m) (*function*), 278
Parametrization(F, D) (*function*), 354
Parent(A) (*function*), 252
Parent(a) (*function*), 22, 83, 254, 320
Parent(F) (*function*), 66, 288
Parent(I) (*function*), 330
Parent(K) (*function*), 9
Parent(O) (*function*), 66, 288
Parent(P) (*function*), 342
Parent(R) (*function*), 265
Parent(w) (*function*), 83
Parent(x) (*function*), 204, 223, 225
Partial Fraction Decomposition (*example*), 270
PartialFractionDecomposition(f) (*function*), 269
PermutationCharacter(K) (*function*), 403
pFundamentalUnits(K, p) (*function*), 100
pFundamentalUnits(O, p) (*function*), 100
PHI1 eq PHI2 (*operation*), 227
Picard Group and Units of an Order (*example*), 219
PicardGroup(O) (*function*), 37, 94
PicardGroup(S) (*function*), 219
PicardNumber(O) (*function*), 37
PicClass(x) (*function*), 225
pIntegralBasis(I, p) (*function*), 136, 370
Place(I) (*function*), 28, 129, 338, 341
PlaceEnumCopy(R) (*function*), 400
PlaceEnumCurrent(R) (*function*), 400
PlaceEnumInit(K) (*function*), 400
PlaceEnumInit(K, Pos) (*function*), 400
PlaceEnumInit(P) (*function*), 400
PlaceEnumNext(R) (*function*), 400
PlaceEnumPosition(R) (*function*), 400
Places(F) (*function*), 289, 340, 346
Places(F, m) (*function*), 312, 341
Places(K) (*function*), 28, 129
PlacesAboveRationalPrime(E, p) (*function*), 220
pMaximalOrder(O, p) (*function*), 60, 287, 334
PoleDivisor(a) (*function*), 374
PoleDivisor(D) (*function*), 349
Poles(a) (*function*), 326, 340
Poles(F, a) (*function*), 326
Polynomial(G) (*function*), 392
PowerBasis(A) (*function*), 206
PowerProduct(B, E) (*function*), 121
PowerProduct(P, E) (*function*), 26, 92
PowerProduct(Q, S) (*function*), 328
PowerSumToElementarySymmetric(I) (*function*), 165
pRadical(O, p) (*function*), 60, 287, 334
Precision(PHI) (*function*), 227
Predicates on OM Divisors (*example*), 377
PrettyPrintInvariant(I) (*function*), 155
Primary(a) (*function*), 40
PrimeField(A) (*function*), 203, 252
PrimeField(F) (*function*), 11, 71, 288
PrimePowerRepresentation(x, k, a) (*function*), 328
PrimeRepresentative(c, f: parameters) (*function*), 95
PrimeRepresentative(I: parameters) (*function*), 95
PrimeRing(A) (*function*), 252
PrimeRing(F) (*function*), 11, 71, 288
PrimeRing(O) (*function*), 71, 288
PrimeRing(R) (*function*), 265
Primes with 2 as a Biquadratic Residue (*example*), 41
PrimesAbove(I) (*function*), 220
PrimitiveElement(A) (*function*), 206
PrimitiveElement(F) (*function*), 86
PrimitiveElement(I) (*function*), 109

PrimitiveElement(K) (*function*), 23, 86
PrimitiveElement(O) (*function*), 86, 292
PrimitiveElementResidueField(P) (*function*), 213
PrincipalDivisor(a) (*function*), 325
PrincipalDivisorMap(F) (*function*), 356
PrincipalIdealMap(O) (*function*), 314
PrintSeqAlgEtQElt(seq) (*function*), 227
PrintWKICM(R) (*function*), 227
Probable Automorphism Group of a Class Field (*example*), 190
ProbableAutomorphismGroup(A) (*function*), 190
Product Representation of Elements (*example*), 328
ProductOfEquationOrders(A) (*function*), 208
ProductRepresentation(a) (*function*), 26, 92, 328
ProductRepresentation(P, E) (*function*), 26, 92
ProductRepresentation(Q, S) (*function*), 328
Prune(A) (*procedure*), 257
pSelmerGroup(p, S) (*function*), 177
Puiseux Expansions over an Algebraic Closure (*example*), 251

Q

QNF() (*function*), 3, 49
Qround(E, M) (*function*), 22, 84
QuadraticCharacter(e) (*function*), 232
QuadraticClassGroupTwoPart(d) (*function*), 38
QuadraticClassGroupTwoPart(K) (*function*), 38
QuadraticClassGroupTwoPart(O) (*function*), 38
QuadraticDefect(a, p) (*function*), 91
QuadraticField(m) (*function*), 34
QuadraticForm(I) (*function*), 41
Quintic Fields with a Given Signature (*example*), 33
quo< FldNum : R | f > (*constructor*), 6
quo< O | I > (*constructor*), 125, 339
quo< O | M > (*constructor*), 125, 339
quo< O | m > (*constructor*), 126
quo< O | p > (*constructor*), 339
quo< O | p, r > (*constructor*), 339
quo< O | S > (*constructor*), 125, 339
Quotient Rings of an Order (*example*), 126
Quotient Rings of Ideals (*example*), 339
Quotient(I, J) (*function*), 213
Quotient(I, zbJ) (*function*), 212
Quotient(S, zbJ) (*function*), 213
QuotientRing(A) (*function*), 253
QuotientVS(I, J, P) (*function*), 213, 214
Quotrem(a, b) (*function*), 126
Quotrem(D, k) (*function*), 346
Quotrem(F, G) (*function*), 392
Quotrem(P, k) (*function*), 343

R

r * x (*operation*), 360
R . i (*operation*), 152, 390
R eq S (*operation*), 390
R meet I (*operation*), 117
RadicalExtension(F, d, a) (*function*), 5, 50
Ramification Groups of a Normal Field (*example*), 144
RamificationDegree(I) (*function*), 110, 337
RamificationDegree(I, p) (*function*), 110
RamificationDegree(P) (*function*), 343
RamificationDivisor(D) (*function*), 353
RamificationDivisor(F) (*function*), 294
RamificationField(p) (*function*), 144
RamificationField(p, i) (*function*), 143
RamificationGroup(p) (*function*), 143
RamificationGroup(p, i) (*function*), 143
RamificationIndex(I) (*function*), 110, 337
RamificationIndex(I, p) (*function*), 110
RamificationIndex(P) (*function*), 30, 131, 220, 343
Random(A) (*function*), 205
Random(A, bd) (*function*), 205
Random(F, m) (*function*), 8, 62, 320
Random(F, n) (*function*), 391
Random(G) (*function*), 230
Random(I) (*function*), 212
Random(I, bd) (*function*), 212
Random(I, m) (*function*), 62
Random(icm) (*function*), 225
Random(L) (*function*), 166
Random(O) (*function*), 208
Random(O, bd) (*function*), 208
Random(O, m) (*function*), 62, 320
Random(OQ) (*function*), 126
Random(W) (*function*), 223, 389
Random(W, n) (*function*), 389
RandomPlace(F, m) (*function*), 313, 341
RandomUnit(A, bd) (*function*), 205
Rank(A) (*function*), 253
Rank(F) (*function*), 265
Rank(R) (*function*), 153
RationalDecomposition(A) (*function*), 405
RationalExtensionRepresentation(F) (*function*), 289
RationalFunction(a) (*function*), 328
RationalFunction(a, R) (*function*), 328
RationalFunctionField(R) (*function*), 263
RationalFunctionField(R, r) (*function*), 263
RationalReconstruction(e, f) (*function*), 328
RationalsAsNumberField() (*function*), 3, 49
RawEval(I, GR) (*function*), 237
Ray Class and Ray Residue Groups (*example*), 382
RayClassField(D) (*function*), 181
RayClassField(m) (*function*), 180
RayClassField(m, I) (*function*), 180
RayClassField(m, I, T) (*function*), 180
RayClassField(P) (*function*), 181
RayClassGroup(D) (*function*), 176, 381
RayClassGroup(I) (*function*), 175

RayClassGroup(I, T) (*function*), 175
RayClassGroup(P) (*function*), 176
RayClassGroupDiscLog(y, D) (*function*), 382
RayResidueRing(D) (*function*), 177, 381
RayResidueRing(I) (*function*), 177
RayResidueRing(I, T) (*function*), 177
RayResidueRing(P) (*function*), 177
RealEmbeddings(a) (*function*), 29, 87
RealPlaces(K) (*function*), 28, 129
RealSigns(a) (*function*), 29
Reconstruct(x, R) (*function*), 127
Reconstruction(L, x) (*function*), 20
ReconstructionEnvironment(p, k) (*function*), 127
Reduce(O) (*function*), 291
ReducedBasis(D) (*function*), 377
ReducedDiscriminant(F) (*function*), 75
ReducedDiscriminant(O) (*function*), 75
Reduction(a, P) (*function*), 137, 372
Reduction(a, P, m) (*function*), 137, 372
Reduction(D) (*function*), 351
Reduction(D, A) (*function*), 351
Reduction(I) (*function*), 42
ReferenceDivisor(F) (*function*), 374
Refining an OM Prime Ideal Approximation (*example*), 367
Regulator(K) (*function*), 15, 76
Regulator(O) (*function*), 314
Regulator(O: parameters) (*function*), 76
RegulatorLowerBound(K) (*function*), 15, 76
RegulatorLowerBound(O) (*function*), 76
RelationMatrix(O) (*function*), 96
Relations(L, R) (*function*), 327, 361
Relations(L, R, m) (*function*), 327, 361
Relations(O) (*function*), 96
RelativeField(F, L) (*function*), 10, 67
RelativeField(Q, L) (*function*), 10, 67
RelativeInvariant(G, H) (*function*), 154
RelativeQuadraticDefect(a, p) (*function*), 91
RelativeResolvent(S, U) (*function*), 155
Rep(L) (*function*), 166
RepresentationMatrix(a) (*function*), 25, 90, 323
RepresentationMatrix(a, R) (*function*), 25, 90, 324
Representative(A) (*function*), 247
Representative(F) (*function*), 264, 319
Representative(K) (*function*), 9, 63
Representative(L) (*function*), 166
Representative(O) (*function*), 63, 319
RepresentativeMap(icm) (*function*), 225
RepresentativeMap(W) (*function*), 223
Representatives(icm) (*function*), 225
Representatives(W) (*function*), 223
Residual Degree and Ramification of OM Primes (*example*), 138
Residue Field and Residue Ring (*example*), 213
Residue(d, P) (*function*), 360
ResidueClassDegree(I) (*function*), 337
ResidueClassDegree(P) (*function*), 343
ResidueClassField(I) (*function*), 110, 338
ResidueClassField(O, I) (*function*), 110
ResidueClassField(P) (*function*), 30, 131, 343
ResidueField(I) (*function*), 138, 373
ResidueField(P) (*function*), 213
ResidueGRH(O, B) (*function*), 96
ResidueGRHbound(O, e) (*function*), 96
ResidueRing(S, I) (*function*), 213
ResidueRingUnits(I) (*function*), 219
ResidueRingUnits(S, I) (*function*), 219
ResidueRingUnitsSubgroupGenerators(F) (*function*), 219
Restrict(chi, D) (*function*), 230
Restrict(chi, I) (*function*), 230
Restrict(chi, I, oo) (*function*), 230
Restrict(G, D) (*function*), 230
Restrict(G, H) (*function*), 230
Restrict(G, I) (*function*), 230
Restrict(G, I, oo) (*function*), 230
Restrict(GR, I) (*function*), 238
Restrict(psi, H) (*function*), 230
Restrict(psi, I) (*function*), 230
Restrict(psi, I, oo) (*function*), 230
Riemann-Roch Space of a Divisor (*example*), 351
RiemannRochSpace(D) (*function*), 351
RiemannRochSpace(P) (*function*), 351
RingClassGroup(O) (*function*), 94
RingOfIntegers(F) (*function*), 8, 35, 57, 265
RingOfIntegers(O) (*function*), 53
Rohrlich's Root Number Exercise (*example*), 244
Root(a, n) (*function*), 22, 84, 248
Root(I, k) (*function*), 118
Root(I, n) (*function*), 331
RootNumber(A) (*function*), 406
RootNumber(A,infty) (*function*), 406
RootNumber(A,p) (*function*), 406
RootNumber(GR) (*function*), 243
RootNumber(GR, P) (*function*), 243
RootNumber(GR, p) (*function*), 243
RootNumber(psi) (*function*), 243
RootNumber(psi, P) (*function*), 243
RootNumber(psi, p) (*function*), 243
RootNumbers(GR) (*function*), 243
RootNumbers(psi) (*function*), 243
RootOfUnity(n) (*function*), 44
RootOfUnity(n, A) (*function*), 248
RootOfUnity(n, K) (*function*), 44
Roots via Order Reconstruction (*example*), 128
Roots(f) (*function*), 248
Roots(f, A) (*function*), 248
Roots(f, D) (*function*), 327

S

S ! I (*operation*), 341

S * gen (*operation*), 210
S eq I (*operation*), 210
S meet I (*operation*), 211
S subset I (*operation*), 211
S1 eq S2 (*operation*), 342
S1 ne S2 (*operation*), 342
S-Units of a Real Quadratic Field (*example*), 123
SafeUniformizer(P) (*function*), 338
SClassGroup(S) (*function*), 357
SClassGroupAbelianInvariants(S) (*function*), 357
SClassGroupExactSequence(S) (*function*), 357
SClassNumber(S) (*function*), 357
SemiReducedBasis(D) (*function*), 377
SeparatingElement(F) (*function*), 294
SerreBound(F) (*function*), 312
SerreBound(F, m) (*function*), 312
SerreBound(q, g) (*function*), 312
SetClassGroupBounds(n) (*procedure*), 98
SetClassGroupBounds(string) (*procedure*), 98
SetEvaluationComparison(R, F, n) (*procedure*), 153
SetKantPrecision(F, n) (*procedure*), 66
SetKantPrecision(F, n, m) (*procedure*), 66
SetKantPrecision(O, n) (*procedure*), 66
SetKantPrecision(O, n, m) (*procedure*), 66
SetKantPrinting(f) (*function*), 66
SetOrderMaximal(O, b) (*procedure*), 83, 284
SetOrderTorsionUnit(O, e, r) (*procedure*), 83
SetOrderUnitsAreFundamental(O) (*procedure*), 83
SetPrintClassGroupWarnings(b) (*procedure*), 93
SetRepresentative(x, J) (*procedure*), 223
SetTargetRing(~ chi, e) (*procedure*), 231
SetTargetRing(~ psi, e) (*procedure*), 231
Setting Class Group Bounds (*example*), 98
SetUseMontes(f) (*procedure*), 133, 367
SetUseMontes(t, f) (*procedure*), 133, 367
SetVerbose("Montes", v) (*procedure*), 133, 368
SetVerbose(s, n) (*procedure*), 64
SFL(P, s) (*procedure*), 133, 367
ShortBasis(D : parameters) (*function*), 350
ShortElement(I) (*function*), 217
Sign(a, p) (*function*), 399
Signature(F) (*function*), 15, 76
Signature(O) (*function*), 76
SimNEQ(K, e, f) (*function*), 104
SimpleExtension(F) (*function*), 10, 67
SimpleExtension(O) (*function*), 67
Simplify(A) (*procedure*), 256
Simplify(O) (*function*), 69, 286
Sine and Cosine via Exponentials (*example*), 352
Single Factor Lifting of an OM Prime (*example*), 133
SingularPrimes(R) (*function*), 220
SIntegralBasis(I, S) (*function*), 136, 370
SLPolynomialRing(R, n) (*function*), 152
SmallRepresentative(I) (*function*), 217
SMaximalOrder(u, S) (*function*), 389
Solutions(t, a) (*function*), 105
SolveByRadicals(f) (*function*), 161, 303
Solving a Polynomial by Radicals (*example*), 162, 304
Solving a Relative Norm Equation (*example*), 27, 104
Solving a Thue Equation (*example*), 105
Solving a Unit Equation (*example*), 106
SpaceOfDifferentialsFirstKind(F) (*function*), 358
SpaceOfHolomorphicDifferentials(F) (*function*), 358
SpecialEvaluate(F, x) (*function*), 392, 393
Splitting Field of a Degree 8 Polynomial (*example*), 259
Splitting Field of a Quartic as a Number Field (*example*), 260
Splitting Field of Degree 720 as a Tower (*example*), 261
SplittingField(F) (*function*), 5, 50
SplittingField(f) (*function*), 6, 50
SplittingField(L) (*function*), 6, 50
SPrincipalDivisorMap(S) (*function*), 357
Sqrt(a) (*function*), 22, 84, 248
Sqrt(I) (*function*), 118, 331
SquarefreePartialFractionDecomposition(f) (*function*), 270
SquareRoot(a) (*function*), 22, 84, 248
SquareRoot(I) (*function*), 118, 331
SRegulator(S) (*function*), 357
Stauduhar(G, H, S, B) (*function*), 150
Strong Approximation (*example*), 398
StrongApproximation(m, S) (*function*), 398
StrongApproximation(S, Z, V) (*function*), 398
sub< D | : parameters> (*constructor*), 31, 365
sub< D | dabs : parameters> (*constructor*), 31
sub< D | dmin, dmax : parameters> (*constructor*), 31
sub< F | e_1, ..., e_n > (*constructor*), 6, 51
sub< F | S > (*constructor*), 6, 51
sub< O | a_1, ..., a_r > (*constructor*), 53
sub< O | f > (*constructor*), 35
sub<F | S> (*constructor*), 281
sub<F | s_1, ..., s_r > (*constructor*), 281
Subfield Lattice of an Octic Field (*example*), 167
Subfield(L, K) (*function*), 19
SubfieldLattice(K) (*function*), 166
Subfields and Isomorphism Testing (*example*), 307
Subfields of a Function Field (*example*), 304
Subfields(F) (*function*), 304
Subfields(K) (*function*), 26, 165
Subfields(K, n) (*function*), 165
SubOrder(O) (*function*), 53, 288

SuccessiveMinima(D) (*function*), 378
SUnitAction(SU, Act, S) (*function*), 124
SUnitCohomologyProcess(S, U) (*function*), 168
SUnitDiscLog(SU, L, S) (*function*), 124
SUnitDiscLog(SU, x, S) (*function*), 124
SUnitGroup(I) (*function*), 122
SUnitGroup(S) (*function*), 122, 356
Support(D) (*function*), 29, 130, 341, 349, 377
Support(I) (*function*), 121
Support(L) (*function*), 121
Support(P) (*function*), 341
Swinnerton-Dyer Polynomials (*example*), 250

T

T !! I (*operation*), 210
TargetRestriction(G, C) (*function*), 231
TargetRestriction(H, C) (*function*), 231
Tate-Lichtenbaum Pairing of Divisors (*example*), 357
TateLichtenbaumPairing(D1, D2, m) (*function*), 357
TateTwist(GR, n) (*function*), 238
TateTwist(psi, n) (*function*), 238
TeichmuellerSystem(R) (*function*), 389
Testing Weak Equivalence (*example*), 224
Testing Whether an Ideal Is a Square (*example*), 331
Three Ways to Create an Element (*example*), 62
ThueSystem(f) (*function*), 105
ThueSystem(O) (*function*), 105
Top(L) (*function*), 166
TorsionUnitGroup(K) (*function*), 99
TorsionUnitGroup(O) (*function*), 99
TotalDegree(f) (*function*), 267
TotallyRealPositiveUnitGroup(S) (*function*), 227
TotallyRealSubAlgebra(K) (*function*), 227
TotallyRealUnitGroup(S) (*function*), 227
TotallyUnitTrivialSubgroup(G) (*function*), 229
Trace(a) (*function*), 24, 89, 255, 323
Trace(a, R) (*function*), 24, 89, 323
Trace(P, a) (*function*), 170
Trace(x) (*function*), 215
TraceAbs(a) (*function*), 24, 89
TraceDualIdeal(I) (*function*), 215
TraceDualIdeal(O) (*function*), 216
TraceMatrix(F) (*function*), 80
TraceMatrix(O) (*function*), 80
TraceMinimalCone(nf) (*function*), 101
TransformationMatrix(I) (*function*), 111, 335
TransformationMatrix(O, P) (*function*), 78
TransformationMatrix(O1, O2) (*function*), 292
TrialDivision(n, B) (*function*), 40
Trivial Cartier Action on Differentials (*example*), 362
Twisted and Additive Polynomials (*example*), 391
TwistedPolynomials(R) (*function*), 390
Twisting a Grossencharacter by a Hilbert Character (*example*), 240
Two-Element Presentation of an Ideal (*example*), 112
Two-Part of a Quadratic Class Group (*example*), 38
TwoCocycle(A) (*function*), 191
TwoElement(I) (*function*), 112, 136, 335, 371
TwoElementNormal(I) (*function*), 112
TwoGeneratingSet(I) (*procedure*), 218
TwoGenerators(P) (*function*), 344
Type of a Sequence (*example*), 273
Type(a) (*function*), 22
Type(K) (*function*), 9
TypeOfSequence(f) (*function*), 272

U

Underlying Rings of a Relative Function Field (*example*), 290
UnderlyingField(F) (*function*), 289
UnderlyingField(F, R) (*function*), 289
UnderlyingRing(F) (*function*), 289
UnderlyingRing(F, R) (*function*), 289
Uniformizers and Completions (*example*), 216
Uniformizers(PPs) (*function*), 221
UniformizingElement(P) (*function*), 30, 109, 131, 344
Unit Group of an Equation Order (*example*), 314
UnitEquation(a, b, c) (*function*), 106
UnitGroup(K) (*function*), 99
UnitGroup(O) (*function*), 99, 313
UnitGroup(OQ) (*function*), 126
UnitGroup(S) (*function*), 219
UnitGroupAsSubgroup(O) (*function*), 100
UnitRank(K) (*function*), 15, 76, 99
UnitRank(O) (*function*), 76, 99, 313
UnitSquareClassReps(p) (*function*), 92
UnitsWithSigns(K, oo, Signs) (*function*), 101
UnitsWithSigns(K, x) (*function*), 101
UnitsWithSigns(O, oo, Signs) (*function*), 101
UnitsWithSigns(O, x) (*function*), 101
UnitsWithSigns(x) (*function*), 101
UnitTrivialSubgroup(G) (*function*), 229

V

v @ a (*operation*), 170
Valuation(a, P) (*function*), 325, 334, 344
Valuation(a, p) (*function*), 29, 130
Valuation(alpha, P : parameters) (*function*), 137, 371
Valuation(D, P) (*function*), 351, 377
Valuation(d, P) (*function*), 360
Valuation(I, P) (*function*), 137, 220, 334, 372
Valuation(I, p) (*function*), 29, 110, 130
Valuation(w, I) (*function*), 92
Valuation(x, P) (*function*), 220
ValuationRing(F) (*function*), 265

ValuationRing(F, f) (*function*), 265
VectorSpace(K, J) (*function*), 13, 73
VectorSpace(K, J, S) (*function*), 13, 73
VerifyRelation(f, F) (*function*), 163
VerschiebungImage(e) (*function*), 389
VerschiebungMap(W) (*function*), 388

W

W ! a (*operation*), 388
W ! x (*operation*), 222
w * v (*operation*), 83
w + v (*operation*), 83
W . 1 (*operation*), 388
w / v (*operation*), 83
w - v (*operation*), 83
w ^ k (*operation*), 83
w div v (*operation*), 83
w eq v (*operation*), 85
w in O (*operation*), 85
w ne v (*operation*), 85
W1 eq W2 (*operation*), 223
Weak Equivalence Classes (*example*), 223
WeakApproximation(I, V) (*function*), 122, 338
WeakEquivalenceClassesWithPrescribed MultiplicatorRing(S) (*function*), 222
WeakEquivalenceClassMonoid(E) (*function*), 222
WeakEquivalenceClassMonoidAbstract(R) (*function*), 222
WEClass(x) (*function*), 225
WeierstrassPlaces(D) (*function*), 353
WeierstrassPlaces(F) (*function*), 294, 342
WeightedDegree(f) (*function*), 267
WeilDescent(E,k) (*function*), 362
WeilDescentDegree(E,k) (*function*), 363
WeilDescentGenus(E,k) (*function*), 363
WeilRestriction(E, n) (*function*), 290
WittRing(F, n) (*function*), 388
WKICM(E) (*function*), 222
WKICM_bar(S) (*function*), 222
WronskianOrders(D) (*function*), 354
WronskianOrders(F) (*function*), 294

X

x * I (*operation*), 115, 211
x * O (*operation*), 107, 330
x * r (*operation*), 360
x * y (*operation*), 153, 223, 225
x + y (*operation*), 153, 360
x / r (*operation*), 360
x / y (*operation*), 360
x @ a (*operation*), 170
x - y (*operation*), 153, 360
x ^ n (*operation*), 205, 223, 225
x eq y (*operation*), 360
x in D (*operation*), 360
x in I (*operation*), 113, 210
x in icm (*operation*), 225
x in O (*operation*), 208
x in W (*operation*), 223, 388
x notin I (*operation*), 113
x1 * x2 (*operation*), 205
x1 + x2 (*operation*), 205
x1 / x2 (*operation*), 205
x1 - x2 (*operation*), 205
x1 eq x2 (*operation*), 205, 223, 225
XGcd(a, b) (*function*), 126

Z

ZBasis(I) (*function*), 210
ZBasis(S) (*function*), 207
ZBasisLLL(~S) (*procedure*), 212
Zero(A) (*function*), 204, 247
Zero(F) (*function*), 264, 319
Zero(K) (*function*), 9, 63
Zero(O) (*function*), 63, 319
Zero(R) (*function*), 390
Zero(S) (*function*), 208
Zero(W) (*function*), 388
ZeroDivisor(a) (*function*), 374
ZeroDivisor(D) (*function*), 349
Zeroes(a) (*function*), 325
Zeroes(F, a) (*function*), 326
Zeroes(F, n) (*function*), 15, 76
Zeroes(O, n) (*function*), 76
Zeros and Defining Polynomial (*example*), 15, 77
Zeros(a) (*function*), 325, 340
Zeros(F, a) (*function*), 326
Zeros(F, n) (*function*), 76
Zeros(O, n) (*function*), 76
ZetaFunction(F) (*function*), 312
ZetaFunction(F, m) (*function*), 312