

Handbook of Magma Functions

Part III: Basic Rings

Version 2.30-1

Sydney

November 10, 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

18 Introduction to Rings **1**
- 18.1 Overview . . . 1
- 18.2 The World of Rings . . . 2
- 18.3 Coercion . . . 2
- 18.4 Generic Ring Functions . . . 4
- 18.5 Generic Element Functions . . . 6
- 18.6 Ideals and Quotient Rings . . . 9
- 18.7 Other Ring Constructions . . . 11

19 Ring of Integers **12**
- 19.1 Introduction . . . 12
- 19.2 Creation Functions . . . 13
- 19.3 Structure Operations . . . 15
- 19.4 Element Operations . . . 16
- 19.5 Random Numbers . . . 20
- 19.6 GCD and LCM . . . 21
- 19.7 Arithmetic Functions . . . 22
- 19.8 Combinatorial Functions . . . 24
- 19.9 Primes and Primality Testing . . . 24
- 19.10 Factorization . . . 28
- 19.11 Factorization Sequences . . . 34
- 19.12 Modular Arithmetic . . . 36
- 19.13 Infinities . . . 37
- 19.14 Advanced Factorization Techniques: The Number Field Sieve . . . 39
- 19.15 Bibliography . . . 48

20 Integer Residue Class Rings **50**
- 20.1 Introduction . . . 50
- 20.2 Ideals of $\mathbb{Z}$ . . . 50
- 20.3 $\mathbb{Z}$ as a Number Field Order . . . 51
- 20.4 Residue Class Rings . . . 51
- 20.5 Elements of Residue Class Rings . . . 54
- 20.6 Ideal Operations . . . 56
- 20.7 The Unit Group . . . 57
- 20.8 Dirichlet Characters . . . 59

21 Rational Field **64**
- 21.1 Introduction . . . 64
- 21.2 Creation Functions . . . 65
- 21.3 Structure Operations . . . 66
- 21.4 Element Operations . . . 68

22 Finite Fields **73**
- 22.1 Introduction . . . 73
- 22.2 Creation Functions . . . 74
- 22.3 Structure Operations . . . 80
- 22.4 Element Operations . . . 83

22.5 Polynomials for Finite Fields . . . 87
22.6 Discrete Logarithms . . . 88
22.7 Permutation Polynomials . . . 90
22.8 Bibliography . . . 92

23 Nearfields **93**
23.1 Introduction . . . 93
23.2 Nearfield Properties . . . 93
23.3 Constructing Nearfields . . . 94
23.4 Operations on Elements . . . 97
23.5 Operations on Nearfields . . . 99
23.6 The Group of Units . . . 99
23.7 Automorphisms . . . 101
23.8 Nearfield Planes . . . 101
23.9 Bibliography . . . 103

24 Univariate Polynomial Rings **105**
24.1 Introduction . . . 105
24.2 Creation Functions . . . 105
24.3 Structure Operations . . . 108
24.4 Element Operations . . . 110
24.5 Common Divisors and Common Multiples . . . 117
24.6 Polynomials over the Integers . . . 118
24.7 Polynomials over Finite Fields . . . 118
24.8 Factorization . . . 119
24.9 Ideals and Quotient Rings . . . 123
24.10 Special Families of Polynomials . . . 125
24.11 Bibliography . . . 126

25 Multivariate Polynomial Rings **128**
25.1 Introduction . . . 128
25.2 Polynomial Rings and Polynomials . . . 128
25.3 Structure Operations . . . 131
25.4 Element Operations . . . 133
25.5 Greatest Common Divisors . . . 142
25.6 Factorization and Irreducibility . . . 143
25.7 Resultants and Discriminants . . . 146
25.8 Polynomials over the Integers . . . 147
25.9 Bibliography . . . 147

26 Real and Complex Fields **148**
26.1 Introduction . . . 148
26.2 Creation Functions . . . 150
26.3 Structure Operations . . . 153
26.4 Element Operations . . . 154
26.5 Transcendental Functions . . . 166
26.6 Elliptic and Modular Functions . . . 171
26.7 Theta Functions . . . 176
26.8 Gamma, Bessel, and Associated Functions . . . 176
26.9 The Hypergeometric Function . . . 178
26.10 Other Special Functions . . . 179
26.11 Summation of Infinite Series . . . 180
26.12 Numerical Integration . . . 180
26.13 Bibliography . . . 184

References **186**

Index **190**

Chapter 18

Introduction to Rings

18.1 Overview

Rings of various kinds form the richest source of algebraic structures in Magma. Tables 1 and 2 list the most important types.

Symbol	Description	Category
Z	ring of integers	RngInt
Z/mZ	ring of residue classes	RngRes
R[x]	univariate poly. ring	RngUPol
F[x]/f(x)	univ. poly. factor ring	RngUPolRes
R[x_1,...,x_m]	multivariate poly. ring	RngMPol
R[[x]]	power series ring	RngSer
O	order in a number field	RngOrd
\Z_p	p-adic ring	RngPad
R_m	local ring	RngLoc
V	valuation ring	RngVal
Q	rational field	FldRat
F_q	finite field	FldFin
F(x_1,...,x_m)	rational function field	FldFun
F((x))	field of Laurent series	FldPow
Q(sqrt(D))	quadratic number field	FldQuad
Q(zeta_n)	cyclotomic number field	FldCyc
Q(alpha)	number field	FldNum
Q_p	p-adic field	FldPad
Q_p(alpha)	local field	FldLoc
R	real field	FldRe
C	complex field	FldCom

The list of rings in Table 1 is not exhaustive, for two reasons. In the first place, some rings have been categorized differently, because their module structure or algebra structure seems pre-eminent; thus matrix rings and finitely presented algebras appear (more or less arbitrarily) in the Part on Algebras, and vector spaces and their generalizations appear in the Module Part. (Also, rings of class functions appear in the Part on Groups.) Furthermore, certain general constructions (such as `sub`) allow the user to define rings that do not appear in the above list, most notably subrings of $\mathbb{Z}$. Looking at the table it may seem that all rings in Magma are commutative and unital. This is not the case (even though it would have made life much easier); since polynomial rings and the like can be defined over any coefficient ring, the matrix rings and finitely presented algebras not listed here that are not generally commutative, allow the construction of non-commutative rings. Furthermore, the `sub` constructor allows the creation of rings without 1; certain functions for the construction of new rings from old ones do not allow such non-unital

coefficient rings. In this Chapter we give an overview of the various types and the relations between them. Moreover, we describe the important principles underlying the rules for coercion of elements of one ring into another. This Chapter also describes the common functions for all types of rings (and their elements), and subsequent Chapters deal with particular categories of rings, as indicated by the table.

18.2 The World of Rings

There are various ways in which to order the families of rings appearing in Table 1. We look at some in this section.

18.2.1 New Rings from Existing Ones

It is important to realize at the outset that to work comfortably with a ring, it should be finitely generated (over some subring); indeed, the only violations of this rule occur for real and complex fields, and p-adic and power series type structures, in which we necessarily have to cope with approximations. All other rings we will label as *exact*. All rings in Table 1 can be obtained from the ring of rational integers $\mathbb{Z}$ by repeated application of a handful of fundamental mathematical constructions. The first such construction is *forming fractions*: the rational field $\mathbb{Q}$ can be obtained as the field of fractions of $\mathbb{Z}$. The second construction is that of forming quotients: in this way the rings $\mathbb{Z}/m\mathbb{Z}$ are obtained from $\mathbb{Z}$. The third important construction is that of *transcendental extension*: by adjoining an element that satisfies no relation over the coefficient ring, a polynomial ring is obtained. An *algebraic extension* can be obtained by a combination of a transcendental extension and a quotient. Finally, *completion* of a ring at a prime leads in general to the rings that were labelled above as not exact. Some other constructions are: tensoring, taking direct products (leading to tuple modules), and taking valuation rings (an operation inverse to taking fields of fractions). Most of these constructions are supported by Magma. In many situations the `quo` and `ext` constructors will perform the quotient and algebraic extension operations, just like `sub` creates sub-structures. Note an important distinction: usually `sub` creates structures of exactly the same type as the original structure—this is precisely why the construction of sub-object does not appear as an important construction for creating new objects in the previous paragraph. Care should be taken not to confuse the mathematical properties of rings (or objects in Magma in general) and the properties of the object that Magma is aware of. For example, if one creates the ring of residue classes $\mathbb{Z}/p\mathbb{Z}$ for a prime number p, using the command `IntegerRing(p)`, the Magma object created is a residue class ring (whose modulus happens to be prime) and *not* a finite field; the functions applicable are the residue class ring functions, and it is, for instance, not possible to create a field extension over this object. If the intention was to create a finite field, the `FiniteField(p)` command should have been used, and for that object it is possible to create a field extension. Similarly, a convenient way of thinking about a number field $K = \mathbb{Q}(\alpha)$ is to regard it as a quotient of the polynomial ring $\mathbb{Q}[X]$ and the ideal generated by the minimal polynomial f of the primitive element α:

$$K = \mathbb{Q}(\alpha) \cong \mathbb{Q}[X]/(f).$$

This is, however, not the way to create number fields in Magma. The quotient ring of a polynomial ring will be an object to which only the generic ring functions apply, whereas to obtain the number field with all the machinery to manipulate it one has to use a command like `NumberField(f)`.

18.3 Coercion

A ring element can often be coerced into a ring other than its parent. The need for this occurs for example when one wants to perform a binary ring operation on elements of different structures, or when an intrinsic function is invoked on elements for which it has not been defined. The basic principle is that such an operation may be performed whenever it makes sense mathematically. Before the operation can be performed however, an element may need to be coerced into some structure in which the operation can legally be performed. There are two types of coercion: *automatic* and *forced* coercion. Automatic coercion occurs when Magma can figure out for itself what the target structure should be, and how elements of the originating structure can be coerced into that structure. In other cases Magma may still be able to perform the coercion, provided the target structure has been specified; for this type of coercion `R ! x` instructs Magma to execute the coercion of element x into ring R. The precise rules

for automatic and forced coercion between rings are explained in the next two subsections. It is good to keep an important general distinction between automatic and forced coercion in mind: whether or not automatic coercion will succeed depends on the originating and the target structure alone, while for forced coercion success may depend on the particular element as well. Thus, integers can be lifted automatically to the rationals if necessary, but conversely, only the integer elements of $\mathbb{Q}$ can be coerced into $\mathbb{Z}$ by using `!`. The subsections below will describe for specific rings R and S in Magma whether or not an element r of R can be lifted automatically or by force into S. Suppose that the unary Magma function `Function` takes elements of the type of S as argument and one is interested in the result of that function when applied to r. If R can be coerced automatically into a unique structure S of the desired type, then `Function(r)` will produce the required result. If R cannot be coerced automatically into such S, but forced coercion on r is possible, then `Function(S ! r)` will yield the desired effect. If, finally, neither automatic nor forced coercion is possible, it may be possible to define a map m from R to S, and then `Function( m(r) )` will give the answer. For example, the function `Order` is defined for elements of residue class rings (among others). But `Order(3)` has no obvious interpretation (and an error will arise) because there is not a unique residue class ring in which this should be evaluated. If a binary operation $\circ : C \times C \to C$ on members C of a category of rings $\mathcal{C}$ is applied to elements r and s of members R and S from $\mathcal{C}$, the same rules for coercion will be used to determine the legality of $r \circ s$. If s can be coerced automatically into R, then this will take place and $r \circ s$ will be evaluated in R; otherwise, if r can be coerced automatically into S, then $r \circ s$ will be evaluated in S. If neither succeeds, then, in certain cases, Magma will try to find an existing *common overstructure* T for R and S, that is, an object T from $\mathcal{C}$ such that $\subset T \supset S$; then both r and s will be coerced into T and the result $t = r \circ s$ will be returned as an element of T. If none of these cases apply, an error results. It may still be possible to evaluate by forced coercion of r into S or s into R, using `(S ! r) o s` or using `r o (R ! s)`.

18.3.1 Automatic Coercion

We will first deal with the easier of the two cases: automatic coercion. A simple demonstration of the desirability of automatic coercion is given by the following example:

```
1 + (1/2);
```

It is obvious that one wants the result to be 3/2: we want to identify the integer 1 with the rational number 1 and perform the addition in $\mathbb{Q}$, that is, we clearly wish to have automatic coercion from $\mathbb{Z}$ to $\mathbb{Q}$. The basic rule for automatic coercion is:

automatic coercion will only take place when there exists a unique target structure and an obvious homomorphism from the parent structure to the target structure

In particular, if one structure is naturally contained in the other (and Magma knows about it!), automatic coercion will take place. (The provision that Magma must know about the embedding is in particular relevant for finite fields and number fields; in these cases it is possible to create subrings, or even isomorphic rings/fields, for which the embedding is not known.) Also, for any ring R there is a natural ring homomorphism $\mathbb{Z} \to R$, hence any integer can be coerced automatically into any ring. See the (pdf) Handbook for a table detailing the rules for automatic coercion. It is appropriate here to make some general remarks only. Integers will be automatically coerced into any ring, and rationals into any ring except a finite field or a residue class ring. Operations involving elements of real or complex fields of different precisions automatically coerce the numbers into the field with the higher precision. Elements of different finite field elements can be operated upon with automatic coercion only if one field is a subset of the other or both fields have been created inside another finite field.

In addition to these rules, general rules apply to polynomial and matrix algebras. The rules for polynomial rings are as follows. An element s from a ring S can be automatically coerced into $R[X_1, \ldots, X_n]$ if either S equals $R[X_1, \ldots, X_i]$ for some $1 \leq i \leq n$, or $S = R$. Note that in the latter case the element s must be an element of the coefficient ring R, and that it is not sufficient for it to be coercible into R. So, for example, we can add an integer and a polynomial over the integers, we can add an element f of $\mathbb{Z}[X]$ and g of $\mathbb{Z}[X, Y]$, but *not* an integer and a polynomial over the rationals. An element s can be coerced automatically in the matrix ring $M_{n,n}(R)$ if it is coercible automatically into the coefficient ring R, in which case s will be identified with the diagonal matrix that has each diagonal entry equal to s. So we can add an integer and a matrix ring element over the rationals, but we cannot automatically add elements of $M_{n,n}(\mathbb{Z})$ and $M_{n,n}(\mathbb{Q})$, nor elements from $M_{2,2}(\mathbb{Z})$ and $M_{3,3}(\mathbb{Z})$.

18.3.2 Forced Coercion

In certain cases where automatic coercion will not take place, one can cast an element into the ring in which the operation should take place. If, for example, one is working in a ring $\mathbb{Z}/p\mathbb{Z}$, and p happens to be prime, it may occur that one wishes to perform some finite field operations on an element in the ring; if F is a finite field of characteristic p an element x of $\mathbb{Z}/p\mathbb{Z}$ can be cast into an element of F using `F ! x;`

See the (pdf) Handbook for a table detailing the rules for non-automatic coercion. It is appropriate here to make some general remarks only. Non-automatic coercion may only apply to certain elements of the domain; thus only those elements of $\mathbb{Q}$ can be coerced into $\mathbb{Z}$ that are in fact integers. Similarly, if an element of a quadratic field with a negative discriminant is to be coerced into a real field then it must correspond to a complex number with no imaginary part. An element from $\mathbb{F}_{p^s}$ can only be coerced into $\mathbb{Z}/m\mathbb{Z}$ if $s = 1$ and $m = p$. Any element from $\mathbb{F}_{p^s}$ can be coerced into $\mathbb{F}_{p^r}$ if s divides r, and otherwise only a subset of the field is coercible. Similarly for cyclotomic fields.

The rules for coercion from and to polynomial rings and matrix rings are as follows. If an attempt is made to forcibly coerce s into $P = R[X_1, \ldots, X_n]$, the following steps are executed successively:

(a) if s is an element of P it remains unchanged;

(b) if s is a sequence, then the zero element of P is returned if s is empty, and if it is non-empty but the elements of the sequence can be coerced into $P[X_1, \ldots, X_{n-1}]$ then the polynomial $\sum_j s[j]X_n^{j-1}$ is returned;

(c) if s can be coerced into the coefficient ring R, then the constant polynomial s is returned;

(d) if s is a polynomial in $R[X_1, \ldots, X_k]$ for some $1 \leq k \leq n$, then it is lifted in the obvious way into P;

(e) if s is a polynomial in $R[X_1, \ldots, X_k]$ for some $k > n$, but constant in the indeterminates $X_{n+1}, \ldots, X_k$, then s is projected down in the obvious way to P.

If none of these steps successfully coerces s into P, an error occurs. The ring element s can be coerced into $M_{n,n}(R)$ if either it can be coerced into R (in which case s will be identified with the diagonal matrix with s on the diagonal), or $s \in S = M_{n,n}(R')$, where R' can be coerced into R. Also a sequence of n^2 elements coercible into R can be coerced into the matrix ring $M_{n,n}(R)$. Elements from a matrix ring $M_{n,n}(R)$ can only be coerced into rings other than a matrix ring if $n = 1$; in that case the usual rules for the coefficient ring R apply.

Note that in some cases it is possible to go from (a subset of) some structure to another in two steps, but not directly: it is possible to go

```
> y := L ! (Q ! x)
```

to coerce a rational element of one number field into another via the rationals. Finally we note that the binary Boolean operator `in` returns true if and only if forced coercion will be successful.

18.4 Generic Ring Functions

The generic functions described in this Chapter apply in principle to every type of ring. For certain rings these are the only applicable functions. The qualification 'in principle' in the first sentence is made because for some classes of rings an algorithm to compute certain of these functions does not exist, or has not been implemented. In that case an error will result.

This general list is provided primarily to avoid duplication of common descriptions. In the following Chapters the generic functions will be listed merely without further description, and the emphasis can be on the functions specific to a particular type of ring.

18.4.1 Related Structures

`Parent(R): Rng -> Pow`

The parent of ring R. Currently this returns the *power structure* of the ring.

`Category(R): Rng -> Cat`

`Type(R): Rng -> Cat`

The 'type' of R, that is, the Magma category to which the ring R belongs. The procedure call `ListCategories()` gives a list of all the categories, as does the Appendix.

`PrimeField(F): Fld -> Fld`

For a field F, this returns either $\mathbb{F}_p$, if the characteristic p of F is positive, or $\mathbb{Q}$, if the characteristic of F is 0. If F is an extension field then it will return the field at the bottom of the extension tower.

`PrimeRing(R): Rng -> Rng`

For a unitary ring R, this returns either $\mathbb{Z}/n\mathbb{Z}$, if the characteristic n of R is positive, or $\mathbb{Z}$, if the characteristic of R is 0. If R is an extension ring then it will return the ring at the bottom of the extension tower.

`Centre(R): Rng -> Rng`

`Center(R): Rng -> Rng`

Given a ring R, return its centre, consisting of the subring of elements commuting with all other elements of R.

18.4.2 Numerical Invariants

`Characteristic(R): Rng -> RngIntElt`

The characteristic of the ring R, which is the smallest positive integer m such that $m \cdot r = 0$ for every $r \in R$, or zero if such m does not exist.

`# R: Rng -> RngIntElt`

The cardinality of the ring R; here R must be finite.

18.4.3 Predicates and Boolean Operations

`IsCommutative(R): Rng -> BoolElt`

Returns `true` if it is known that the ring R is commutative, `false` if it is known that R is not commutative. An error results if the answer is not known.

`IsUnitary(R): Rng -> BoolElt`

Returns `true` if the ring R is known to be unitary (that is, if R has a multiplicative identity), `false` if R has no 1.

`IsFinite(R): Rng -> BoolElt`

Returns `true` if the ring R is known to be a finite ring, `false` if it is known to be infinite. An error results if the answer is not known.

`IsOrdered(R): Rng -> BoolElt`

Returns `true` if the ring R has a total ordering defined on the set of its elements, `false` otherwise.

`IsField(R): Rng -> BoolElt`

Returns `true` if the ring R is known to be a field, `false` if it is known to not be a field. An error results if the answer is not known.

`IsDivisionRing(R): Rng -> BoolElt`

Returns `true` if the ring R is known to be a division ring (that is, every non-zero element is invertible), `false` if it is known that R is not a division ring. An error results if the answer is not known.

`IsEuclideanDomain(R): Rng -> BoolElt`

Returns `true` if the ring R is known to be a euclidean domain, `false` if it is known that R is not a euclidean domain. An error results if the answer is not known.

`IsEuclideanRing(R): Rng -> BoolElt`

Returns `true` if the ring R is known to be euclidean, `false` if it is known that R is not euclidean. An error results if the answer is not known.

`IsMagmaEuclideanRing(R): Rng -> BoolElt`

Returns `true` iff the ring R is a computable euclidean ring within Magma (i.e., iff the necessary euclidean operations are defined for R so algorithms requiring a euclidean ring will work).

`IsPID(R): Rng -> BoolElt`

`IsPrincipalIdealDomain(R): Rng -> BoolElt`

Returns `true` if the ring R is known to be a principal ideal domain, `false` if it is known that R is not a principal ideal domain. An error results if the answer is not known.

`IsPIR(R): Rng -> BoolElt`

`IsPrincipalIdealRing(R): Rng -> BoolElt`

Returns `true` if the ring R is known to be a principal ideal ring, `false` if it is known that R has non-principal ideals. An error results if the answer is not known.

`IsUFD(R): Rng -> BoolElt`

`IsUniqueFactorizationDomain(R): Rng -> BoolElt`

Returns `true` if the ring R is known to be a unique factorization domain, `false` if it is known that R is not a unique factorization domain. An error results if the answer is not known.

`IsDomain(R): Rng -> BoolElt`

`IsIntegralDomain(R): Rng -> BoolElt`

Returns `true` if it is known that R is an integral domain (i.e., R has no zero divisors), `false` if R is known to have zero divisors. An error results if the answer is not known.

`HasGCD(R): Rng -> BoolElt`

Returns `true` iff there is a GCD algorithm for elements of ring R in Magma.

`R eq S: Rng, Rng -> Rng`

For certain pairs R, S of rings, this returns `true` if R and S refer to the same ring, and `false` otherwise. However, if R and S belong to different categories an error may result.

`R ne S: Rng, Rng -> Rng`

For certain pairs R, S of rings, this returns `true` if R and S refer to different rings, and `false` otherwise. However, if R and S belong to different categories an error may result.

18.5 Generic Element Functions

18.5.1 Parent and Category

`Parent(r): RngElt -> Rng`

The (default) parent ring of ring element r. Usually the parent of r has been created explicitly before, but in certain cases, such as literal integers, rationals, reals, and values returned by certain functions a default parent is created in the background.

`Category(r): RngElt -> Cat`

`Type(r): RngElt -> Cat`

The 'type' of r, that is, the Magma category to which the ring element r belongs. The procedure call `ListCategories()` gives a list of all the categories, as does the Appendix.

18.5.2 Creation of Elements

`Zero(R): Rng -> RngElt`

The zero element of ring R; this is equivalent to `R ! 0`.

`One(R): Rng -> RngElt`

`Id(R): Rng -> RngElt`

The multiplicative identity 1 of ring R; this is equivalent to `R ! 1`.

`R ! a: Rng, RngElt -> RngElt`

Coerce the element a of some ring into the ring R. (The rules on coercion are explained earlier in this Chapter.) If a is an integer, the coercion will always succeed: the element $a \cdot 1_R$ will be returned, where 1_R is the unit element of R.

`Random(R): Rng -> RngElt`

A random element of the finite ring R (every element of R has the same probability of being returned).

`Representative(R): Rng -> RngElt`

`Rep(R): Rng -> RngElt`

A representative element of the finite ring R.

18.5.3 Arithmetic Operations

`+ a: RngElt -> RngElt`

Element a.

`- a: RngElt -> RngElt`

The negation (additive inverse) of element a.

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

The sum of the ring elements a and b; if a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the sum can be taken.

`a - b: RngElt, RngElt -> RngElt`

The difference of the ring elements a and b; if a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the difference can be taken.

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

The product of the ring elements a and b; if a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the product can be taken.

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

Form the k-th power of the ring element a, for small k. If $k = 0$, then a^k returns 1, even when $a = 0$. If k is negative, a must have a multiplicative inverse and the result is $(a^{-1})^{-k}$.

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

Given an element a of R and a unit b of R, form the quotient of the elements a and b. If b is not invertible in R, an error results, unless both a and b are integers, in which case `a / b` returns the rational number a/b. If a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the quotient can be taken.

`a +:= b: RngElt, RngElt -> RngElt`

Mutation assignment: change a into the sum of a and b.

`a -:= b: RngElt, RngElt -> RngElt`

Mutation assignment: change a into the difference of a and b.

`a *:= b: RngElt, RngElt -> RngElt`

Mutation assignment: change a into the product of a and b.

`a /:= b: RngElt, RngElt -> RngElt`

Mutation assignment: change a into the quotient of a and b.

`a ^:= k: RngElt, RngIntElt -> RngElt`

Mutation assignment: change a into the power a^k.

18.5.4 Equality and Membership

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

Returns `true` if the elements a and b of R are the same, otherwise `false`.

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

Returns `true` if the elements a and b of R are distinct, otherwise `false`.

`R eq S: Rng, Rng -> BoolElt`

Returns `true` if the rings R and S are the same, otherwise `false`.

`R ne S: Rng, Rng -> BoolElt`

Returns `true` if the rings R and S are distinct, otherwise `false`.

`a in R: RngElt, Rng -> BoolElt`

Returns `true` if and only if a is an element of R.

`a notin R: RngElt, Rng -> BoolElt`

Returns `true` if and only if a is not an element of R.

18.5.5 Predicates on Ring Elements

`IsZero(a): RngElt -> BoolElt`

Returns `true` if and only if the element a of R equals 0_R.

`IsOne(a): RngElt -> BoolElt`

Returns `true` if and only if the element a of R equals 1_R.

`IsMinusOne(a): RngElt -> BoolElt`

Returns `true` if and only if the element a of R equals the element -1 of R.

`IsUnit(a): RngElt -> BoolElt`

Returns `true` if a is a unit in its parent R, `false` otherwise.

`IsIdempotent(x): RngElt -> BoolElt`

Returns `true` if and only if x^2 equals x.

`IsNilpotent(x): RngElt -> BoolElt`

Returns `true` if and only if some integer power x^i of x is zero.

`IsZeroDivisor(x): RngElt -> BoolElt`

Returns `true` if and only if x is a zero-divisor, that is, there exists an element y in the parent R of x such that $xy = 0$.

`IsIrreducible(x): RngElt -> BoolElt`

Returns `true` if and only if the parent R of the element x is a domain and x is irreducible in R, that is, x is a non-unit of R and whenever a product ab of elements of R divides x then a or b is a unit of R.

`IsPrime(x): RngElt -> BoolElt`

Returns `true` if and only if the parent R of the element x is a domain and x is a prime element of R, that is, x is neither 0 nor a unit and whenever x divides the product ab of two elements of R it divides a or b.

18.5.6 Comparison of Ring Elements

The comparison operations are only defined on types of ring that are ordered.

`a gt b: RngElt, RngElt -> BoolElt`

Returns `true` if the ring element a is greater than the ring element b, otherwise `false`.

`a ge b: RngElt, RngElt -> BoolElt`

Returns `true` if the ring element a is greater than or equal to the ring element b, otherwise `false`.

`a lt b: RngElt, RngElt -> BoolElt`

Returns `true` if the ring element a is less than the ring element b, otherwise `false`.

`a le b: RngElt, RngElt -> BoolElt`

Returns `true` if the ring element a is less than or equal to the ring element b, otherwise `false`.

`Maximum(a, b): RngElt, RngElt -> RngElt`

The maximum of the ring elements a and b; if a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the maximum can be taken.

`Maximum(Q): [RngIntElt] -> RngElt`

The maximum of the sequence Q of ring elements.

`Minimum(a, b): RngElt, RngElt -> RngElt`

The minimum of the ring elements a and b; if a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the minimum can be taken.

`Minimum(Q): [RngIntElt] -> RngElt`

The minimum of the sequence Q of ring elements.

18.6 Ideals and Quotient Rings

The following entries describe the operations on ideals in a *commutative ring* R. Certain operations on left and right ideals in non-commutative rings will be described in the Chapters for the corresponding rings.

18.6.1 Defining Ideals and Quotient Rings

`ideal< R | a1, ..., ar >: Rng, RngElt, ..., RngElt -> RngIdl`

Given a ring R and elements $a_1, \ldots, a_r$ of R, create the ideal I of R generated by $a_1, \ldots, a_r$.

`quo< R | ar, ..., ar >: Rng, RngElt, ..., RngElt -> Rng`

Given a ring R and elements $a_1, \ldots, a_r$ of R, construct the quotient ring $Q = R/I$, where I is the ideal of R generated by $a_1, \ldots, a_r$.

`R / I: Rng, RngIdl -> Rng`

Given a ring R and an ideal I of R, construct the quotient ring $Q = R/I$, as well as the canonical map $R \to R/I$.

Note, however, that there can be pitfalls with this, particularly if the first argument is itself interpreted as an ideal, when instead the result could be interpreted as something like the `ColonIdeal`.

For instance:

```
> Z := Integers();
> I := ideal<Z|1>; // ideal of Z
> Z/I; // ideal quotient, similar to ColonIdeal
Integer Ring
> quo<Z|I>; // quotient of ring by ideal
Residue class ring of integers modulo 1
```

In fact, $\mathbb{Z}/I$ is computing the (fractional) ideal J such that $JI = \mathbb{Z}$. The technical reason for this is that both $\mathbb{Z}$ and I are of type `RngInt`, and there is no specific `RngIntIdl` type as with order of number fields (where there are `RngOrd` and `RngOrdIdl`).

On the other hand, for instance for univariate polynomial rings where again there is no specific type for ideals, the `/` operator acts as the ring/ideal quotient (while `ColonIdeal` does not exist for this type, though it does for the multivariate polynomial ring type `RngMPol`).

```
> Zx<x> := PolynomialRing(Rationals());
> I := ideal<Zx|x>;
> Type(Zx),Type(I);
RngUPol RngUPol
> Zx/I; // same as quo<Zx|I>
Univariate Quotient Polynomial Algebra in over Rational Field
> Type(Zx/I);
RngUPolRes
```

`PowerIdeal(R): Rng -> PowIdl`

The set of ideals of R. This is the parent of all ideals of R.

18.6.2 Arithmetic Operations on Ideals

`I + J: RngIdl, RngIdl -> RngIdl`

The sum of the ideals I and J of the ring R. This ideal consists of elements $a + b$, with $a \in I$ and $b \in J$. If I is generated by $\{a_1, \ldots, a_k\}$ and J is generated by $\{b_1, \ldots, b_m\}$, then $I + J$ is generated by $\{a_1, \ldots, a_k, b_1, \ldots, b_m\}$.

`I * J: RngIdl, RngIdl -> RngIdl`

The product of the ideals I and J of the ring R. This is the ideal generated by elements $a \cdot b$, with $a \in I$ and $b \in J$, and it consists of elements $a_1b_1 + \cdots + a_nb_n$, with $a_i \in I$ and $b_j \in J$.

`I meet J: RngIdl, RngIdl -> RngIdl`

The intersection of the ideals I and J of the ring R.

18.6.3 Boolean Operators on Ideals

Throughout this subsection I and J are ideals belonging to the same integer ring R, while a is an element of R.

`a in I: RngElt, RngIdl -> BoolElt`

Returns `true` if and only if the element a is a member of the ideal I.

`a notin I: RngElt, RngIdl -> BoolElt`

Returns `true` if and only if the element a is not a member of the ideal I.

`I eq J: RngIdl, RngIdl -> BoolElt`

Returns `true` if and only if the ideals I and J are equal.

`I ne J: RngIdl, RngIdl -> BoolElt`

Returns `true` if and only if the ideals I and J are distinct.

`I subset J: RngIdl, RngIdl -> BoolElt`

Returns `true` if and only if the ideal I is contained in the ideal J.

`I notsubset J: RngIdl, RngIdl -> BoolElt`

Returns `true` if and only if the ideal I is not contained in the ideal J.

18.7 Other Ring Constructions

Magma allows the construction of residue fields, localization of rings, and completion of rings. These constructions really just create appropriate rings of different categories within Magma.

18.7.1 Residue Class Fields

`ResidueClassField(I): Rng -> Fld, Map`

Given a maximal ideal I of a ring R, create the residue class field K of the quotient ring R/I, together with a map sending an element of R to the corresponding element of K.

18.7.2 Localization

`loc< R | a`$_1$`, ..., a`$_r$` >: Rng, RngElt, ..., RngElt -> Rng, Map`

Given a ring R and elements $a_1, \ldots, a_r$ of R, which generate a prime ideal P of R, create the localization L of R at P, together with a map sending an element of R to the corresponding element of L.

`Localization(R, P): Rng, Rng -> Rng, Map`

Given a ring R and a prime ideal P of R, create the localization L of R at P, together with a map sending an element of R to the corresponding element of L.

18.7.3 Completion

`comp< R | a`$_1$`, ..., a`$_r$` >: Rng, RngElt, ..., RngElt -> Rng, Map`

Given a ring R and elements $a_1, \ldots, a_r$ of R, which generate a prime ideal or zero ideal P of R, create the completion C of R at P, together with a map sending an element of R to the corresponding element of C.

`Completion(R, P): Rng, Rng -> Rng, Map`

Given a ring R and a prime ideal or zero ideal P of R, create the completion C of R at P, together with a map sending an element of R to the corresponding element of C.

18.7.4 Transcendental Extension

`ext< R | >: Rng -> RngUPol`

Given a ring R create the univariate transcendental extension $R[x]$ of R. This is equivalent to `PolynomialRing(R)`.

`ext< R, n | >: Rng, RngIntElt -> RngMPol`

Given a ring R and an integer $n \geq 1$, create the multivariate transcendental extension $R[x_1, \ldots, x_n]$ of R. This is equivalent to `PolynomialRing(R, n)`.

Chapter 19

Ring of Integers

19.1 Introduction

This Chapter describes the operators and functions for working with the ring of rational integers $\mathbb{Z}$. Integers are the most commonly used objects in Magma. They can be created by just typing in the literal (decimal) digits. Integers thus created are elements of the ring of integers which is automatically created when Magma is started up. There is just one single object 'integer ring' around, but references to it (new 'names' for it) can be created using the `IntegerRing` function.

19.1.1 Representation

Since large integers occur so frequently, the first requirement for a computer algebra system is to support fast arithmetic for integers of arbitrary size. Indeed, within the bounds set by the available memory, it is possible to operate reasonably efficiently with integers of any number of decimal digits. Although it is well possible to use the integer facilities without being aware of the internal representation of (large) integers, it is sometimes useful to know how integers are stored. The most important fact is that integers smaller than $2^{30} = 1073741824$ in absolute value are 'single precision', and in many circumstances such 'small integers' allow considerably faster arithmetic (they are treated slightly differently internally and escape the overhead of memory management used to deal with multi-precision integers).

19.1.2 Coercion

Integers will be automatically coerced into almost every unitary ring R using the identification of 1 and 1_R. This means that integer arguments are allowed for almost any ring element function, and that it is not necessary to convert an integer before applying binary operators (such as $+$) on a combination of arguments consisting of an integer and another ring element. For more on coercion we refer to Chapter *Introduction to Rings*.

19.1.3 Homomorphisms

Ring homomorphisms are required to be unitary. Therefore, to specify a homomorphism with the integers as its domain requires merely the specification of the codomain.

`hom< Z -> R | >: RngInt, Rng -> Map`

The natural homomorphism from $\mathbb{Z}$ to the ring R.

Example: Homomorphism from the Integers to a Matrix Ring (ex-a58966)

```
> h := hom< Integers() -> MatrixRing(RealField(12), 3) | >;
> h(2)^-1;
[0.5   0   0]
[  0 0.5   0]
[  0   0 0.5]
```

19.2 Creation Functions

19.2.1 Creation of Structures

The ring of integers is automatically created when Magma is first loaded. The ring may be formally created (and, if desired, assigned to a variable) using the function `IntegerRing()`. Subrings of $\mathbb{Z}$ are always ideals; see the section on ideals for details.

`IntegerRing() -> RngInt`

`Integers() -> RngInt`

`RingOfIntegers() -> RngInt`

`IntegerRing(Q): FldRat -> RngInt`

`Integers(Q): FldRat -> RngInt`

`RingOfIntegers(Q): FldRat -> RngInt`

Create the ring of integers $\mathbb{Z}$.

19.2.2 Creation of Elements

Since the ring of integers is present when Magma is started up, integers typed into Magma without any explicit context will be regarded as elements of the ring of integers. Integers can be specified using decimal, hexadecimal, and binary notation.

`a`$_1$`a`$_2$`...a`$_r$

Given a succession of decimal digits $a_1, \ldots, a_r$, create the corresponding integer. Leading zeros will be ignored.

`0xa`$_1$`a`$_2$`...a`$_r$

Given a succession of hexadecimal digits $a_1, \ldots, a_r$, create the corresponding integer. Leading zeros will be ignored.

`0ba`$_1$`a`$_2$`...a`$_r$

Given a succession of binary digits $a_1, \ldots, a_r$, create the corresponding integer. Leading zeros will be ignored.

`elt< Z | a`$_1$`a`$_2$`...a`$_r$ `>: RngInt, RngIntElt -> RngIntElt`

Given a succession of decimal digits $a_1, \ldots, a_r$, create the corresponding integer as an element of Z.

`elt< Z | 0xa`$_1$`a`$_2$`...a`$_r$ `>: RngInt, RngIntElt -> RngIntElt`

Given a succession of hexadecimal digits $a_1, \ldots, a_r$, create the corresponding integer as an element of Z.

`elt< Z | 0ba`$_1$`a`$_2$`...a`$_r$ `>: RngInt, RngIntElt -> RngIntElt`

Given a succession of binary digits $a_1, \ldots, a_r$, create the corresponding integer as an element of Z.

`Z ! a: RngInt, RngElt -> RngIntElt`

`Z ! [a]: RngInt, [RngElt] -> RngIntElt`

Coerce the ring element a into the ring of integers Z. The element a is allowed to be an element of the ring of integers modulo m (in which case the result r satisfies $0 \leq r < m$), or an element of a finite field (in which case the result r satisfies $0 \leq r < p$ if a is in the prime field, of characteristic p, and an error otherwise), or an element of the integers, rationals, a quadratic field, a cyclotomic field, or a number field (in which cases the result is the obvious integer if a is integral and an error otherwise).

Example: Creating Integers (ex-2fa3b0)

```
> Z := IntegerRing();
> n := 1234567890;
> n in Z;
```

(continues on next page)

(continued from previous page)

```
true
> m := elt< Z | 1234567890 >;
> m eq n;
true
> l := Z ! elt< QuadraticField(3) | 1234567890, 0>;
> l;
1234567890
> k := elt< Z | 0x499602D2 >;
1234567890
> b := 0b10010011001011000000001011010010;
> b eq n;
true
> c := elt< Z | 0b10010011001011000000001011010010 >;
> c eq n;
true
```

`One(Z): RngInt -> RngIntElt`

`Identity(Z): RngInt -> RngIntElt`

`Zero(Z): RngInt -> RngIntElt`

`Representative(Z): RngInt -> RngIntElt`

These generic functions (cf. Chapter *Introduction to Rings*) create 1, 1, 0, and 0 respectively, in the integer ring $\mathbb{Z}$.

19.2.3 Printing of Elements

Magma supports the printing of integers in both decimal and hexadecimal form. The default print method is to print integers in base 10; base 16 printing is performed using the `Hex` print level.

Example: Printing in Hexadecimal `(ex-47136f)`

```
> n := 1234567890;
> n;
1234567890
> n:Hex;
0x499602D2
```

19.2.4 Element Conversions

`FactorizationToInteger(s): [ <RngIntElt, RngIntElt> ] -> RngIntElt`

`FactorisationToInteger(s): [ <RngIntElt, RngIntElt> ] -> RngIntElt`

`Facint(s): [ <RngIntElt, RngIntElt> ] -> RngIntElt`

Given a sequence of two-element tuples $s = [< p_1, k_1 >, ..., < p_r, k_r >]$ containing pairs of integers $< p_i, k_i >$, $1 \leq i \leq r$, with k_i non-negative, this function returns the integer $p_1^{k_1} \cdots p_r^{k_r}$. It is normally used for converting a factorization sequence to the corresponding integer.

`IntegerToSequence(n, b): RngIntElt, RngIntElt -> [RngIntElt]`

`Intseq(n, b): RngIntElt, RngIntElt -> [RngIntElt]`

Given a non-negative integer n and a positive integer $b \geq 2$, return the unique base b representation of n in the form of a sequence Q. That is, if $n = a_0 b^0 + a_1 b^1 + \ldots + a_{k-1} b^{k-1}$ with $0 \leq a_i < b$ and $a_{k-1} > 0$, then $Q = [a_0, a_1, \ldots, a_{k-1}]$. (If $n = 0$, then $Q = [\,]$.)

`SequenceToInteger(s, b): [RngIntElt], RngIntElt -> RngIntElt`

`Seqint(s, b): [RngIntElt], RngIntElt -> RngIntElt`

Given a positive integer $b \geq 2$ and a sequence $Q = [a_0, \ldots, a_{k-1}]$ of non-negative integers such that $0 \leq a_i < b$, return the integer $n = a_0 b^0 + a_1 b^1 + \ldots + a_{k-1} b^{k-1}$. If Q is the empty sequence, the integer zero is returned. This function performs the inverse operation of the base b representation.

`IntegerToString(n): RngIntElt -> ModStgElt`

Create the string consisting of the decimal digits of the integer n. In the case in which n is negative the first character will be the minus sign.

`IntegerToString(n, b): RngIntElt, RngIntElt -> ModStgElt`

Create the string consisting of the digits of the integer n in base b. In the case in which n is negative the first character will be the minus sign. The base b can be between 2 and 36. For $b \leq 10$, the digits are represented numerically. For $b > 10$, the digits are represented both numerically and alphabetically, so that, 10 is 'A', 11 is 'B', et cetera.

`Eltseq(n): RngIntElt -> [RngIntElt]`

The sequence $[n]$ which can be coerced back into $\mathbb{Z}$.

`Denominator(n): RngIntElt -> RngIntElt`

The denominator of n, i.e., 1.

19.3 Structure Operations

The following generic ring functions are applicable to the ring of integers and its elements.

19.3.1 Related Structures

`Category(Z): RngInt -> Cat`

`Parent(Z): RngInt -> PowerStructure`

`PrimeRing(Z): RngInt -> RngInt`

`Center(Z): RngInt -> RngInt`

`AdditiveGroup(Z): RngInt -> GrpAb, Map`

Create the abelian group of integers under addition. This returns an infinite (additive) abelian group A of rank 1 together with a map from A to the ring of integers Z, sending $A.1$ to 1.

`MultiplicativeGroup(Z): RngInt -> GrpAb, Map`

`UnitGroup(Z): RngInt -> GrpAb, Map`

Create the abelian group of invertible integers, that is, an abelian group isomorphic to the multiplicative subgroup $\langle -1 \rangle$. This returns an (additive) abelian group A of order 2 together with a map from A to the ring of integers Z, sending $A.1$ to -1.

`ClassGroup(Z): RngInt -> GrpAb, Map`

The class group of the ring of $\mathbb{Z}$ (which is trivial).

`FieldOfFractions(Z): RngInt -> FldRat`

Create the field of fractions $\mathbb{Q}$ of the ring of rational integers.

`sub< Z | n >: RngInt, RngIntElt -> RngInt`

Given Z, the ring of integers or an ideal of it, and an element n of Z, create the ideal $aZ \cap Z$ of the ring of integers. Note that this creates an ideal, not just a subring.

19.3.2 Numerical Invariants

`Characteristic(Z): RngInt -> RngIntElt`

`Signature(Z): RngInt -> RngIntElt, RngIntElt`

The signature of $\mathbb{Z}$ as an order of $\mathbb{Q}$, i.e. `1, 0`.

19.3.3 Ring Predicates and Booleans

`IsCommutative(Z): RngInt -> BoolElt`

`IsUnitary(Z): RngInt -> BoolElt`

```
IsFinite(Z): RngInt -> BoolElt

IsOrdered(Z): RngInt -> BoolElt

IsField(Z): RngInt -> BoolElt

IsEuclideanDomain(Z): RngInt -> BoolElt

IsPID(Z): RngInt -> BoolElt

IsUFD(Z): RngInt -> BoolElt

IsDivisionRing(Z): RngInt -> BoolElt

IsEuclideanRing(Z): RngInt -> BoolElt

IsPrincipalIdealRing(Z): RngInt -> BoolElt

IsDomain(Z): RngInt -> BoolElt

Z eq R: RngInt, Rng -> BoolElt

Z ne R: RngInt, Rng -> BoolElt
```

19.4 Element Operations

19.4.1 Arithmetic Operations

Magma includes both the Karatsuba algorithm and the Schönhage-Strassen FFT-based algorithm for the multiplication of integers ([AHU74, Chap. 7], [vzGG99, Sec. 8.3]). The crossover point (where the FFT method beats the Karatsuba method) is currently 2^{15} bits (approx. 10000 decimal digits) on Sun SPARC workstations and 2^{17} bits (approx. 40000 decimal digits) on Digital Alpha workstations. Assembler macros are used for critical operations and 64-bit operations are used on DEC-Alpha machines.

Magma also contains an asymptotically-fast integer (and polynomial) division algorithm which reduces division to multiplication with a constant scale factor that is in the practical range. Thus division of integers and polynomials are based on the Karatsuba and Schönhage-Strassen (FFT) methods when applicable. The crossover point for integer division (when the new method outperforms the classical method) is currently at the point of dividing a 2^{12} bit (approx. 1200 decimal digit) integer by a 2^{11} (approx. 600 decimal digit) integer on Sun SPARC workstations.

```
+ n: RngIntElt -> RngIntElt
- n: RngIntElt -> RngIntElt
m + n: RngIntElt, RngIntElt -> RngIntElt
m - n: RngIntElt, RngIntElt -> RngIntElt
m * n: RngIntElt, RngIntElt -> RngIntElt
n ^ k: RngIntElt, RngIntElt -> RngIntElt
m / n: RngIntElt, RngIntElt -> RngIntElt
m +:= n: RngIntElt, RngIntElt -> RngIntElt
m -:= n: RngIntElt, RngIntElt -> RngIntElt
m *:= n: RngIntElt, RngIntElt -> RngIntElt
m /:= n: RngIntElt, RngIntElt -> RngIntElt
m ^:= k: RngIntElt, RngIntElt -> RngIntElt
n div m: RngIntElt, RngIntElt -> RngIntElt
```

The quotient q of the division with remainder $n = qm + r$, where $0 \leq r < m$ or $m < r \leq 0$ (depending on the sign of m), for integers n and $m \neq 0$.

`n mod m: RngIntElt, RngIntElt -> RngIntElt`

The remainder r of the division with remainder $n = qm + r$, where $0 \leq r < m$ or $m < r \leq 0$ (depending on the sign of m), for integers n and $m \neq 0$.

`ExactQuotient(n, d): RngIntElt, RngIntElt -> RngIntElt`

Assuming that the integer n is exactly divisible by the integer d, return the exact quotient of n by d (as an integer). An error results if d does not divide n exactly.

`n div:= m: RngIntElt, RngIntElt -> RngIntElt`

`n mod:= m: RngIntElt, RngIntElt -> RngIntElt`

19.4.2 Bit Operations

The following functions use bit operations on the internal representation, so are in general quicker than using the usual arithmetic operators.

`ShiftLeft(n, b): RngIntElt, RngIntElt -> RngIntElt`

Given integers n and b, with $b \geq 0$, return $n \times 2^b$.

`ShiftRight(n, b): RngIntElt, RngIntElt -> RngIntElt`

Given integers n and b, with $b \geq 0$, return n `div` 2^b.

`ModByPowerOf2(n, b): RngIntElt, RngIntElt -> RngIntElt`

Given integers n and b, with $b \geq 0$, return $n \bmod 2^b$ (so the result is always non-negative).

19.4.3 Bitwise Operations

The following functions apply logical operators to integers by interpreting them as sequences of bits. Note that the (implicitly infinite) 2-adic expansion of these integers is used, which enables consistent handling of negative integers. In particular, the `BitwiseNot` of a non-negative integer will be negative, and vice versa.

`BitwiseNot(x): RngIntElt -> RngIntElt`

The integer whose 2-adic representation is the bitwise 'not' of the 2-adic representation of x.

`BitwiseAnd(x, y): RngIntElt, RngIntElt -> RngIntElt`

The integer whose 2-adic representation is the bitwise 'and' of the 2-adic representations of x and y.

`BitwiseOr(x, y): RngIntElt, RngIntElt -> RngIntElt`

The integer whose 2-adic representation is the bitwise 'or' of the 2-adic representations of x and y.

`BitwiseXor(x, y): RngIntElt, RngIntElt -> RngIntElt`

The integer whose 2-adic representation is the bitwise 'xor' of the 2-adic representations of x and y.

19.4.4 Equality and Membership

`m eq n: RngIntElt, RngIntElt -> BoolElt`

`m ne n: RngIntElt, RngIntElt -> BoolElt`

`n in R: RngIntElt, Rng -> BoolElt`

`n notin R: RngIntElt, Rng -> BoolElt`

19.4.5 Parent and Category

`Parent(n): RngIntElt -> RngInt`

`Category(n): RngIntElt -> Cat`

19.4.6 Predicates on Ring Elements

`IsEven(n): RngIntElt -> BoolElt`

Returns `true` if the integer n is even, otherwise `false`.

`IsOdd(n): RngIntElt -> BoolElt`

Returns `true` if the integer n is odd, otherwise `false`.

`IsDivisibleBy(n, d): RngIntElt, RngIntElt -> BoolElt, RngIntElt`

Returns `true` if and only if the integer n is divisible by the integer d; if `true`, the quotient of n by d is also returned.

`IsSquare(n): RngIntElt -> BoolElt, RngIntElt`

Returns `true` if the non-negative integer n is the square of an integer, `false` otherwise. If n is a square, its positive square root is also returned.

`IsSquarefree(n): RngIntElt -> BoolElt`

Returns `true` if the non-zero integer n is not divisible by the square of any prime, `false` otherwise.

`IsPower(n): RngIntElt -> BoolElt`

If the integer $n > 1$ is a power $n = b^k$ of an integer b, with $k > 1$, this function returns true, the minimal positive b and its associated k; if it is not such integer power the function returns false.

`IsPower(n, k): RngIntElt, RngIntElt -> BoolElt`

If the integer $n > 1$ is k-th power, with $k > 1$, of some integer b, so that $n = b^k$, this function returns true, and b; if it is not a k-th integer power the function returns false.

`IsPrime(n): RngIntElt -> BoolElt`

`Proof: BoolElt` Default: `true`

Returns `true` if and only if the integer n is a prime. A rigorous primality test which returns a proven result will be used unless the parameter `Proof` is `false`. The reader is referred to Section *Primes and Primality Testing* for a complete description of this function.

Example: Primes Congruent to 3 Modulo 4 (ex-687a13)

In this example we find some 10-digit primes that are congruent to 3 modulo 4 such that $(p-1)/2$ is also prime.

```
> { p : p in [10^10+3..10^10+1000 by 4] |
>        IsPrime(p) and IsPrime((p-1) div 2)};
{ 10000000259, 10000000643 }
```

`IsIntegral(n): RngIntElt -> BoolElt`

Returns `true` if and only if a is integral, which is of course true for every integer n.

`IsSinglePrecision(n): RngIntElt -> BoolElt`

Returns `true` if n fits in a single word in the internal representation of integers in Magma, that is, if $|n| < 2^{30}$, `false` otherwise.

`IsZero(n): RngIntElt -> BoolElt`

`IsOne(n): RngIntElt -> BoolElt`

`IsMinusOne(n): RngIntElt -> BoolElt`

`IsNilpotent(n): RngIntElt -> BoolElt`

`IsIdempotent(n): RngIntElt -> BoolElt`

`IsUnit(n): RngIntElt -> BoolElt`

`IsZeroDivisor(n): RngIntElt -> BoolElt`

`IsRegular(n): RngInt -> BoolElt`

`IsIrreducible(n): RngIntElt -> BoolElt`

`IsPrime(n): RngIntElt -> BoolElt`

19.4.7 Comparison of Ring Elements

`m gt n: RngIntElt, RngIntElt -> BoolElt`

`m ge n: RngIntElt, RngIntElt -> BoolElt`

`m lt n: RngIntElt, RngIntElt -> BoolElt`

`m le n: RngIntElt, RngIntElt -> BoolElt`

`Maximum(m, n): RngIntElt, RngIntElt -> RngIntElt`

`Maximum(Q): [RngIntElt] -> RngIntElt`

`Minimum(m, n): RngIntElt, RngIntElt -> RngIntElt`

`Minimum(Q): [RngIntElt] -> RngIntElt`

19.4.8 Conjugates, Norm, and Trace

`ComplexConjugate(n): RngIntElt -> RngIntElt`
The complex conjugate of n, which will be the integer n itself.

`Conjugate(n): RngIntElt -> RngIntElt`
The conjugate of n, which will be the integer n itself.

`Norm(n): RngIntElt -> RngIntElt`
The norm in $\mathbb{Q}$ of n, which will be the integer n itself.

`EuclideanNorm(n): RngIntElt -> RngIntElt`
The Euclidean norm (length) of n, which will equal the absolute value of n.

`Trace(n): RngIntElt -> RngIntElt`
The trace (in $\mathbb{Q}$) of n, which will be the integer n itself.

`MinimalPolynomial(n): RngIntElt -> RngUPolElt`
Returns the minimal polynomial of the integer n, which is the monic linear polynomial with constant coefficient n in a univariate polynomial ring R over the integers.

19.4.9 Other Elementary Functions

`AbsoluteValue(n): RngIntElt -> RngIntElt`

`Abs(n): RngIntElt -> RngIntElt`
Absolute value of the integer n.

`Ilog2(n): RngIntElt -> RngIntElt`
The integral part of the logarithm to the base two of the positive integer n.

`Ilog(b, n): RngIntElt, RngIntElt -> RngIntElt`

The integral part of the logarithm to the base b of the positive integer n i.e., the largest integer k such that $b^k \leq n$. The integer b must be greater than or equal to two.

`Quotrem(m, n): RngIntElt, RngIntElt -> RngIntElt, RngIntElt`

Returns both the quotient q and remainder r obtained upon dividing the integer m by the integer n, that is, $m = q \cdot n + r$, where $0 \leq r < n$ if $n > 0$ and $n < r \leq 0$ if $n < 0$.

`Valuation(x, p): RngIntElt, RngIntElt -> RngIntElt, RngIntElt`

The valuation of the integer x at the prime p. This is the largest integer v for which p^v divides x. If $x = 0$ then $v = \infty$. The optional second return value is the integer u such that $x = p^v u$.

`Iroot(a, n): RngIntElt, RngIntElt -> RngIntElt`

Given a positive integer a, return the integer $b = \lfloor \sqrt[n]{a} \rfloor$, i.e. the integral part of the n-th root of a. To obtain the actual root (as a real number), a must e coerced into a real field and the function `Root` applied.

`Sign(n): RngIntElt -> RngIntElt`

Returns -1, 0, or 1 depending upon whether the integer n is negative, zero, or positive, respectively.

`Ceiling(n): RngIntElt -> RngIntElt`

The ceiling of the integer n, that is, n itself.

`Floor(n): RngIntElt -> RngIntElt`

The floor of the integer n, that is, n itself.

`Round(n): RngIntElt -> RngIntElt`

This function rounds the integer n to itself.

`Truncate(n): RngIntElt -> RngIntElt`

This function returns the integer truncation of the integer n, that is, n itself.

`SquarefreeFactorization(n): RngIntElt -> RngIntElt, RngIntElt`

Given a non-negative integer n, return a squarefree integer x as well as a positive integer y, such that $n = xy^2$.

`Isqrt(n): RngIntElt -> RngIntElt`

Given a positive integer n, return the integer $\lfloor \sqrt{n} \rfloor$, i.e., the integral part of the square root of the integer n.

19.5 Random Numbers

Pseudo-random integers in Magma are generated using the *Monster* random number generator of G. Marsaglia [Mar00] (see also [MZ91] and [Mar92]). The period of the Monster generator is $2^{29430} - 2^{27382}$ (approximately 10^{8859}), and the generator passes all of the stringent tests in Marsaglia's *Diehard* test suite [Mar95]. Throughout the following text, the word 'random' is used to mean 'pseudo-random'.

`Random(a, b): RngIntElt, RngIntElt -> RngIntElt`

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

`Random(b): RngIntElt -> RngIntElt`

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

`RandomBits(n): RngIntElt -> RngIntElt`

A random integer m such that $0 \leq m < 2^n$, where n is a small non-negative integer. Thus, m has n random bits with a probability of $1/2$ for each bit. The function always returns 0 when $n = 0$.

`RandomPrime(n: parameter): RngIntElt -> RngIntElt`

`Proof: BoolElt` `Default: true`

A random prime integer m such that $0 < m < 2^n$, where n is a small non-negative integer. The function always returns 0 for $n = 0$ or $n = 1$. A rigorous method will be used to check primality, unless $m > 331704406467988738596 1981$ and the optional parameter `Proof` is set to `Proof := false`, in which case the result indicates that m is a probable prime (of order 20).

`RandomPrime(n, a, b, x: parameter): RngIntElt, RngIntElt, RngIntElt -> BoolElt, RngIntElt`

`Proof: BoolElt` `Default: true`

Tries up to x iterations to find a random prime integer m congruent to a modulo b such that $0 < m < 2^n$. If successful, the function returns true and the integer m, otherwise false. The integer n must be larger than 0, a must lie between 0 and $b - 1$ and b must be larger than 0. A rigorous method will be used to establish primality, unless $m > 331704406467988738596 1981$ and the optional parameter `Proof` is set to `Proof := false`, in which case the result indicates that m is a probable prime (of order 20).

`RandomConsecutiveBits(n, a, b): RngIntElt, RngIntElt -> RngIntElt`

A integer m such that $0 \leq m < 2^n$, and the binary expansion of n consists of consecutive strings of zeros or ones each of random length in the range $[a \ldots b]$.

19.6 GCD and LCM

Within the classical range, Magma uses the fast classical Accelerated GCD algorithm of Kenneth Weber [Web95] to compute the GCD of two integers, and the fast classical Lehmer extended GCD ('XGCD') algorithm [Knu97, pp. 345--348] (which is about 5 times faster than the Euclidean XGCD algorithm) to compute the extended GCD of two integers.

For larger integers, Magma uses the asymptotically fast Schönhage recursive ("half-GCD") algorithm ([Sch71]; see also [Mon92, Sec. 3.8] for the basic idea, applied to polynomials). On a Sun SPARC workstation, the crossover point for the Schönhage GCD algorithm (where it beats the classical Accelerated GCD algorithm) is 32768 bits (about 10000 decimal digits), while the crossover point for the Schönhage XGCD algorithm (when it beats the Lehmer XGCD algorithm) is 6000 bits (about 2000 decimal digits).

`GreatestCommonDivisor(m, n): RngIntElt, RngIntElt -> RngIntElt`

`Gcd(m, n): RngIntElt, RngIntElt -> RngIntElt`

`GCD(m, n): RngIntElt, RngIntElt -> RngIntElt`

The greatest common divisor of m and n, normalized to be non-negative. If either of the inputs is zero, then the result is the absolute value of the other input, while if m and n are both zero the result is zero.

`GreatestCommonDivisor(s): [RngIntElt] -> RngIntElt`

`Gcd(s): [RngIntElt] -> RngIntElt`

`GCD(s): [RngIntElt] -> RngIntElt`

The GCD of the entries of the sequence s. If all entries of the sequence are zero, the result is zero. An error results if the sequence is the null sequence.

`ExtendedGreatestCommonDivisor(m, n): RngIntElt, RngIntElt -> RngIntElt, RngIntElt, RngIntElt`

`Xgcd(m, n): RngIntElt, RngIntElt -> RngIntElt, RngIntElt, RngIntElt`

`XGCD(m, n): RngIntElt, RngIntElt -> RngIntElt, RngIntElt, RngIntElt`

The extended GCD of m and n; returns integers g, x, and y such that g is the greatest common divisor of the integers m and n, and $g = x \cdot m + y \cdot n$. If m and n are both zero, g is zero; otherwise g is always positive. If m and n are both non-zero, the multipliers x and y are unique.

`ExtendedGreatestCommonDivisor(s): [RngIntElt] -> RngIntElt, [RngIntElt]`

`Xgcd(s): [RngIntElt] -> RngIntElt, [RngIntElt]`

`XGCD(s): [RngIntElt] -> RngIntElt, [RngIntElt]`

Given a sequence of integers $s = [s_1, \ldots, s_r]$, return the non-negative integer g and a sequence $X = (x_1, \ldots, x_r)$ such that g is the greatest common divisor of the integers s_i and $g = \sum_{i=1}^{r} x_i \cdot s_i$.

`LeastCommonMultiple(m, n): RngIntElt, RngIntElt -> RngIntElt`

`Lcm(m, n): RngIntElt, RngIntElt -> RngIntElt`

`LCM(m, n): RngIntElt, RngIntElt -> RngIntElt`

The smallest non-negative integer divisible by both m and n. If m or n equals zero, the result is zero; this ensures that $\mathrm{lcm}(m, n)\gcd(m, n) = m \cdot n$.

`LeastCommonMultiple(s): [RngIntElt] -> RngIntElt`

`Lcm(s): [RngIntElt] -> RngIntElt`

`LCM(s): [RngIntElt] -> RngIntElt`

Least common multiple of the sequence of integers s.

19.7 Arithmetic Functions

Each of the functions in this section may take an integer or the factorization of that integer.

`CarmichaelLambda(n): RngIntElt -> RngIntElt`

`CarmichaelLambda(Q): RngIntEltFact -> RngIntElt`

`CarmichaelLambda(Q): [Tup] -> RngIntElt`

The Carmichael function $\lambda(n)$; its value equals the exponent of $(\mathbb{Z}/n\mathbb{Z})^*$.

`DickmanRho(u): FldReElt -> FldReElt`

Computes $\rho(u)$ where ρ is Dickman's rho function.

`FactoredCarmichaelLambda(n): RngIntElt -> RngIntEltFact`

`FactoredCarmichaelLambda(Q): RngIntEltFact -> RngIntEltFact`

`FactoredCarmichaelLambda(Q): [Tup] -> RngIntEltFact`

The Carmichael function $\lambda(n)$, returned as a factorization sequence.

`DivisorSigma(i, n): RngIntElt, RngIntElt -> RngIntElt`

`DivisorSigma(i, Q): RngIntElt, RngIntEltFact -> RngIntElt`

The divisor function $\sigma_i(n) = \sum_{d|n} d^i$ for integer n and small non-negative integer i.

`NumberOfDivisors(n): RngIntElt -> RngIntElt`

`NumberOfDivisors(Q): RngIntEltFact -> RngIntElt`

The number of divisors of the positive integer n. This is a special case of `DivisorSigma`.

`SumOfDivisors(n): RngIntElt -> RngIntElt`

`SumOfDivisors(Q): RngIntEltFact -> RngIntElt`

The sum of the divisors of the positive integer n. This is a special case of `DivisorSigma`.

`EulerPhi(n): RngIntElt -> RngIntElt`

`EulerPhi(Q): RngIntEltFact -> RngIntElt`

`EulerPhi(Q): [Tup] -> RngIntElt`

The Euler totient function $\phi(n)$; its value equals the order of $(\mathbb{Z}/n\mathbb{Z})^*$.

`FactoredEulerPhi(n): RngIntElt -> RngIntEltFact`

`FactoredEulerPhi(Q): RngIntEltFact -> RngIntEltFact`

`FactoredEulerPhi(Q): [Tup] -> RngIntEltFact`

The Euler totient function $\phi(n)$, returned as a factorization sequence.

`EulerPhiInverse(m): RngIntElt -> RngIntElt`

`EulerPhiInverse(Q): RngIntEltFact -> RngIntElt`

The inverse of the Euler totient function $\phi(n)$; that is, the sorted sequence of all integers n such that $\phi(n) = m$.

`FactoredEulerPhiInverse(n): RngIntElt -> RngIntEltFact`

`FactoredEulerPhiInverse(Q): RngIntEltFact -> RngIntEltFact`

The factored inverse of the Euler totient function $\phi(n)$; that is, the sorted sequence of the factorizations of all integers n such that $\phi(n) = m$.

`LegendreSymbol(n, m): RngIntElt, RngIntElt -> RngIntElt`

The Legendre symbol $\left(\frac{n}{m}\right)$: for prime m this checks whether or not n is a quadratic residue modulo m. The function returns 0 if m divides n, -1 if n is not a quadratic residue, and 1 if n is a quadratic residue modulo m. A fast probabilistic primality test is performed on m. If m fails the test (and is therefore composite), an error results; if it passes the test the Jacobi symbol is computed.

`JacobiSymbol(n, m): RngIntElt, RngIntElt -> RngIntElt`

The Jacobi symbol $\left(\frac{n}{m}\right)$. For odd $m > 1$ this is defined (but not calculated!) as the product of the Legendre symbols $\left(\frac{n}{p_i}\right)$, where the product is taken over all primes p_i dividing m including multiplicities. Quadratic reciprocity is used to calculate this symbol, which has the values -1, 0, or 1.

`KroneckerSymbol(n, m): RngIntElt, RngIntElt -> RngIntElt`

The Kronecker symbol $\left(\frac{n}{m}\right)$. This is the extension of the Jacobi symbol to all integers m, by multiplicativity, and by defining $\left(\frac{n}{2}\right) = (-1)^{(n^2-1)/8}$ for odd n (and 0 for even n) and $\left(\frac{n}{-1}\right) = \pm 1$ according to the sign of n for $n \neq 0$ (and 1 for $n = 0$).

`MoebiusMu(n): RngIntElt -> RngIntElt`

`MoebiusMu(Q): RngIntEltFact -> RngIntElt`

The Möbius function $\mu(n)$. This is a multiplicative function characterized by $\mu(1) = 1$, $\mu(p) = -1$, and $\mu(p^k) = 0$ for $k \geq 2$, where p is a prime number.

Example: Amicable Pairs of Integers (ex-233f46)

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

```
> d := func< m | DivisorSigma(1, m)-m >;
> z := func< m | d(d(m)) eq m >;
> for m := 2 to 10000 do
>     if z(m) then
>         m, d(m);
>     end if;
> end for;
6 6
28 28
220 284
284 220
496 496
1184 1210
1210 1184
2620 2924
2924 2620
5020 5564
5564 5020
6232 6368
6368 6232
8128 8128
```

19.8 Combinatorial Functions

`Binomial(n, r): RngIntElt, RngIntElt -> RngIntElt`

The binomial coefficient $\binom{n}{r}$.

`Multinomial(n, [r1, ... rn]): RngIntElt, [RngIntElt] -> RngIntElt`

Given a sequence $Q = [r_1, \ldots, r_k]$ of positive integers such that $n = r_1 + ... + r_k$, return the multinomial coefficient $\binom{n}{r_1,\ldots,r_k}$.

`Factorial(n): RngIntElt -> RngIntElt`

The factorial $n!$ for positive small integer n.

`IsFactorial(n): RngIntElt -> BoolElt, RngIntElt`

Tests if $n = k!$ for some k. If so, return `true` and k, `false` otherwise.

`Partitions(n): RngIntElt -> [ [ RngIntElt ] ]`

The unrestricted partitions of the positive integer n. This function returns a sequence of integer sequences, each of which is a different sequence of positive integers (in descending order) adding up to n. The integer n must be small.

`NumberOfPartitions(n): RngIntElt -> RngIntElt`

The number of unrestricted partitions of the non-negative integer n. The integer n must be small.

`RestrictedPartitions(n, M): RngIntElt, SetEnum -> [ [ RngIntElt ] ]`

The partitions of the positive integer n, restricted to elements of the positive integer sequence M.

`RestrictedPartitions(n, k, M): RngIntElt, RngIntElt, SetEnum -> [ [ RngIntElt ] ]`

The partitions of the positive integer n into k parts, restricted to elements of the positive integer sequence M.

`StirlingFirst(n, k): RngIntElt, RngIntElt -> RngIntElt`

The Stirling number of the first type, $s(n,k) = \left[{n \atop k}\right]$, where n and k are non-negative integers.

`StirlingSecond(n, k): RngIntElt, RngIntElt -> RngIntElt`

The Stirling number of the second type, $S(n,k) = \left\{{n \atop k}\right\}$, where n and k are non-negative integers.

`Bell(n): RngIntElt -> RngIntElt`

The nth Bell number, giving the number of partitions of a set of size n. (Not to be confused with `NumberOfPartitions(n)`, which gives the number of partitions of the *integer* n.) This is equal to the sum of `StirlingSecond(n,k)` for k between 0 and n (inclusive).

`Fibonacci(n): RngIntElt -> RngIntElt`

Given an integer n, this function returns the n-th Fibonacci number F_n, which can be defined via the recursion $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all integers n. Note that n is allowed to be negative, and that $F_{-n} = (-1)^{n+1}F_n$.

`Lucas(n): RngIntElt -> RngIntElt`

Given an integer n, this function returns the n-th Lucas number L_n, which can be defined via the recursion $L_0 = 2$, $L_1 = 1$, and $L_n = L_{n-1} + L_{n-2}$ for all integers n. Note that n is allowed to be negative, and that $L_{-n} = (-1)^n L_n$.

`GeneralizedFibonacciNumber(g0, g1, n): RngIntElt, RngIntElt, RngIntElt -> RngIntElt`

The nth member of the generalized Fibonacci sequence defined by $G_0 = g_0$, $G_1 = g_1$, and $G_n = G_{n-1} + G_{n-2}$ for all integers n. Note that n is allowed to be negative. The Fibonacci and Lucas numbers are special cases where $(g_0, g_1) = (0, 1)$ or $(2, 1)$ respectively.

19.9 Primes and Primality Testing

Primality testing algorithms enable the user to certify the primality of prime integers. Proving the primality of very big integers can be time consuming and therefore in some of the algorithms using

primes and factorization of integers the user can speed up the algorithm by explicitly allowing Magma to use probable primes rather than certified primes. A *probable prime* is an integer that has failed some compositeness test; if an integer passes a compositeness test it will be composite, but there is a (small) probability that a composite number will fail the test and is hence called a probable prime. Each Miller-Rabin test for instance, has a probability of less than 1/4 of declaring a composite number probably prime; in practice that means that numbers that fail several such cheap independent Miller-Rabin compositeness tests will be prime. Unless specifically asked otherwise, Magma will use rigorous primality proofs.

19.9.1 Primality

If a positive integer n is composite, this can be shown quickly by exhibiting a *witness* to this fact. A witness for the compositeness of n is an integer $1 < a < n$ with the property that

$$a^r \not\equiv 1 \bmod n \quad \text{and} \quad a^{r2^i} \not\equiv -1 \bmod n \text{ for } i = 0, 1, \ldots, k-1$$

where r odd, and k are such that $n - 1 = r \cdot 2^k$. A witness never falsely claims that n is composite, because for prime n it must hold that $a^{n-1} \equiv 1 \bmod n$ and only ± 1 are square roots of 1 modulo prime n. Moreover, it has been shown that a fraction of at least 3/4 of all a in the range $2 \ldots n-1$ will witness the compositeness of n. Thus randomly choosing a will usually quickly expose compositeness. Unless more than 1/4 of all possibilities for a are checked though (which in practice will be impossible for reasonable n) the procedure of checking k bases a at random for being a witness (often referred to as 'Miller-Rabin') will not suffice to prove the primality of n; it does however lend credibility to the claim that n is most likely prime if among k (say 20) random choices for a no witness for compositeness has been found. In such cases n is called *probably prime of order* k, and in some sense the probability that n is composite is less than 4^{-k}. A slight adaptation of this compositeness test can be used for primality proofs in a bounded range. There are no composites smaller than the Sorenson/Webster bound $B = 3317044064679887385961981$ (approx 3.3×10^{24}) for which a witness does not exist among $a = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37$ ([SW17]). Using these values of a for candidate witnesses it is certain that for any number n less than B, the test will either find a witness or correctly declare n prime. Since V2.29, Magma uses this test to prove primality (deterministically) of any prime $n < B$.

But even for large integers it is thus usually easy to identify composites without finding a factor; to be certain that a large probable prime is truly prime, a primality proving algorithm is invoked. Magma uses the ECPP (Elliptic Curve Primality Proving) method. Since V2.29, Magma uses a new implementation, that can handle 1000 digit numbers in a few minutes.

This ECPP method is both fast and rigorous, but for large integers (of say more than 100 decimal digits) it will be still be much slower than the Miller-Rabin compositeness test. The method is too involved to be explained here; we refer the reader to the literature ([AM93], [FKMW04]). The `IsPrime` function invokes ECPP, unless a Boolean flag is used to indicate that only 'probable primality' is required. The latter is equivalent to a call to `IsProbablePrime`.

`IsPrime(n): RngIntElt -> BoolElt`

`IsPrime(n: parameter): RngIntElt -> BoolElt`

`Proof: BoolElt`	`Default: true`

Returns `true` iff the integer n is prime. A rigorous deterministic method will be used, unless $n > B$, where $B = 3317044064679887385961981$ (approx 3.3×10^{24}), and optional parameter `Proof` is set to `Proof := false`, in which case the result indicates that n is a probable prime (a strong pseudoprime to 20 bases).

`SetVerbose("ECPP", v): MonStgElt, Elt`

Sets the verbose level for output when the ECPP algorithm is used in the above primality tests. The legal values are `true`, `false`, 0, 1, and 2 (`false` and `true` are the same as 0 and 1 respectively). Level 1 outputs only basic information about the times for the top-level stages (downrun and uprun). Level 2 outputs full information about every step : this level is very verbose!

`PrimalityCertificate(n): RngIntElt -> SeqEnum`

`CheckCertificate(cert): SeqEnum -> BoolElt`

```
Print: BoolElt                    Default: false
Full : BoolElt                    Default: true
```

`PrimalityCertificate` is a variant on `IsPrime` which uses ECPP and outputs a certificate of primality at the conclusion. If the number n is actually proven to be composite or the test fails, then a runtime error occurs. The certificate is a `Magma` sequence of lists. Each list represents one step of the proof.

The first entry in each list L indicates the method used. If the first enty is `E`, the elliptic curve `E := EllipticCurve(L[6])` is used. Its reduction modulo `n := L[2]` has the point P with coordinates `L[7]`. The step assumes that `p := L[5]` is a prime. In this case P has order p. As the prime `p` is sufficiently large, the primality of `n` follows. The next entry in the certificate will contain a proof that `p` is in fact prime. The elliptic curve was found via the CM approach with discriminant `L[3]` and the number of points on its reduction modulo `n` is given by `L[4]`.

If the first enty is `N-1`, the multiplicative group modulo `n := L[2]` has the element `g := L[4]`. The step assmes that `p := L[3]` is a prime. The `p`-th power of `g` is one modulo `n`. As the prime `p` is sufficiently large, the primality of `n` follows. The next entry in the certificate will contain a proof that `p` is in fact prime.

If the first enty is `Det`, `L[2]` is small enough to be covered by a deterministic Miller-Rabin test.

To confirm that a primality certiciate is correct, the function `CheckCertificate` can be used. It returns `true` if the certificate is corret and `false` otherwise. Setting `Print` to `true` displays the proof in a human readable format. Further, setting `full` to `true` results in an independet check of the result obtained by the deterministic Miller-Rabin test. A skeptical user is advised to check every certificate with this function, as it uses an independent implementation of elliptic curve arithmetic and also replaces the deterministic Miller-Rabin test with an independent test.

`OldCertificate(cert): SeqEnum -> List`

The function `OldCertificate` converts the currenly used primality certificate and returns it in the format used by Magma V 2.28 and earlier. It can be used for an independet check of the certificate by a program that supports that format.

`IsProbablePrime(n: parameter): RngIntElt -> BoolElt`

`IsProbablyPrime(n: parameter): RngIntElt -> BoolElt`

```
Bases: RngIntElt                  Default: 20
```

Returns `true` if and only if the integer n is a probable prime. More precisely, the function returns `true` if and only if either n is prime for $n < 3317044064679887385961981$, or n is a strong pseudoprime for 20 random bases b with $1 < b < n$. By setting the optional parameter `Bases` to some value B, the number of random bases used is B instead of 20.

`IsPrimePower(n): RngIntElt -> BoolElt, RngIntElt, RngIntElt`

Returns `true` if and only if the integer n is a prime power; that is, if n equals p^k for some prime p and exponent $k \geq 1$. If this is the case, the prime p and the exponent k are also returned, Note that the primality of p is rigorously proven.

Example: Probable Repunit Primes (ex-ca5938)

This piece of code uses 5 Miller-Rabin tests to find the next probable *repunit*-prime (consisting of all 1's as decimal digits), using the fact that primes of this form consist of a prime number of digits:

```
> NextPPRepunit := function(nn)
>    n := nn;
>    repeat
>       n := NextPrime(n);
```

(continues on next page)

(continued from previous page)

```
>    until IsProbablePrime( (10^n-1) div 9 : Bases := 5);
>    return n;
> end function;
```

The first few cases are easy:

```
> NextPPRepunit(1);
2
> NextPPRepunit(2);
19
> NextPPRepunit(19);
23
> NextPPRepunit(23);
317
```

So we found a 317 digit prime (although we should check genuine primality, using `IsPrime`)! We leave it to the reader to find the next (it has more than 1000 decimal digits).

19.9.2 Other Functions Relating to Primes

The functions `NextPrime` and `PreviousPrime` can be used to find primes in the neighbourhood of a given integer. After sieving out only multiples of very small primes, the remaining integers are tested for primality in order. Again, a rigorous method is used unless the user flags that probable primes suffice. The `PrimeDivisors` function is different from all other functions in this section since it requires the factorization of its argument.

`NextPrime(n): RngIntElt -> RngIntElt`

`NextPrime(n: parameter): RngIntElt -> RngIntElt`

```
Proof: BoolElt                    Default: true
```

The least prime number greater than n, where n is a non-negative integer. The primality is proved. The optional boolean parameter 'Proof' (`Proof := true` by default) can be set to `Proof := false`, to indicate that the next probable prime (of order 20) may be returned.

`PreviousPrime(n): RngIntElt -> RngIntElt`

`PreviousPrime(n: parameter): RngIntElt -> RngIntElt`

```
Proof: BoolElt                    Default: true
```

The greatest prime number less than n, where $n \geq 3$ is an integer. The primality is proved. The optional boolean parameter 'Proof' (`Proof := true` by default) can be set to `Proof := false`, to indicate that the previous probable prime (of order 20) may be returned.

`PrimesUpTo(B): RngIntElt -> [RngIntElt]`

This function lists the primes up to (and including) the (positive) bound B. The algorithm is not super-optimised, but is reasonable. The limit on B is $2^{30} - 1$ for practical reasons, as otherwise the memory requirements can become too large with non-small integers (see below).

`PrimesInInterval(b, e): RngIntElt, RngIntElt -> [RngIntElt]`

This function lists the primes in the interval from b to e, including the endpoints. The algorithm is not very optimised. The interval must be smaller than 2^{30} in size, but if there are non-small integers ($> 2^{30}$) involved then these will create a large memory overhead in any case. For instance, the `PrimesUpTo` intrinsic for $2^{30} - 1$ returns 54400028 primes in 650 megabytes, while the `PrimesInInterval` intrinsic for $(2^{30} + 1, 2^{31} - 1)$ takes over 5 gigabytes for its array of 50697537 primes.

`NthPrime(n): RngIntElt -> RngIntElt`

Given a number n, this function returns the nth prime. This is implemented for primes up to 10^{10}, or $n \leq 455052511$.

`RandomPrime(n: parameter): RngIntElt -> RngIntElt`

Proof: BoolElt	Default: true

A random prime integer m such that $0 < m < 2^n$, where n is a small non-negative integer. The function always returns 0 for $n = 0$ or $n = 1$. A rigorous method will be used to check primality, unless $m > 331704406467988738596 1981$ and the optional parameter 'Proof' is set to `Proof := false`, in which case the result indicates that m is a probable prime (of order 20).

`RandomPrime(n, a, b, x: parameter): RngIntElt, RngIntElt, RngIntElt -> BoolElt, RngIntElt`

Proof: BoolElt	Default: true

Tries up to x iterations to find a random prime integer m congruent to a modulo b such that $0 < m < 2^n$. Returns true, m if found, or false if not found. n must be larger than 0. a must be between 0 and $b - 1$ and b must be larger than 0. A rigorous method will be used to check primality, unless $m > 331704406467988738596 1981$ and the optional parameter 'Proof' is set to `Proof := false`, in which case the result indicates that m is a probable prime (of order 20).

`PrimeBasis(n): RngIntElt -> [RngIntElt]`

`PrimeDivisors(n): RngIntElt -> [RngIntElt]`

A sequence containing the distinct prime divisors of the positive integer $|n|$.

19.10 Factorization

This section contains a description of most of the machinery provided in Magma for the factorization of integers. An account of the Number Field Sieve is deferred until later in the chapter. In the first subsection the general-purpose `Factorization` function is described. It employs a combination of methods in an attempt to find the complete prime factorization of a given integer. Some control is possible over each of the methods, but in general default choices for the parameters would give good results for a wide range of arguments. In the second subsection we describe functions that enable access to each of the factorization methods available in Magma. The user has control over parameters for these methods. Factorization functions in Magma return a *factorization sequence*. This is a sequence of two-element tuples $[< p_1, k_1 >, \ldots, < p_r, k_r >]$, with $p_1 < p_2 < \cdots < p_r$ distinct prime numbers and k_i positive, which is used to represent integers in factored form: $n = \prod_{i=1}^{r} p_i^{k_i}$. Although such sequences are printed like ordinary sequences, they form a separate category `RngIntEltFact`. Operations on such factorization sequences are described in the next section.

19.10.1 General Factorization

The general `Factorization` function is designed to give close to optimal performance for the factorization of integers that may be encountered in the course of daily computations. The strategy employed is as follows (the next subsection gives a more detailed description of the individual methods). First of all a compositeness test is used to ensure that the argument is composite; if not the primality proving algorithm is invoked (unless a flag is set to avoid this — see below). See the previous section for compositeness testing and primality proving. This operation is repeated for any non-trivial factor (and cofactor) found along the way. Before any of the general factorization techniques is employed, it is checked whether $|n|$ is of the special form $b^k \pm 1$, in which case an intelligent database look-up is used which is likely to be successful if b and k are not too large. This is equivalent to the `Cunningham` function on $b, k, \pm 1$, described in the next subsection. In the first true stage of factorization *trial division* is used to find powers of 2 and other small primes (by default up to 10000). After this it is checked whether the remaining composite number is the power of a positive integer; if so the appropriate root is used henceforth. After this *Pollard's ρ method* is applied (using 8191 iterations by default). The

bound on trial division factors and the number of iterations for ρ can be set by the optional parameters `TrialDivisionLimit` and `PollardRhoLimit`. It is possible, from this point on, that several composite factors still need factorization. The description below applies to each of these. The final two algorithms deployed are usually indicated by ECM (for Elliptic Curve Method) and MPQS (for Multiple Polynomial Quadratic Sieve). By default, ECM (which is likely to find 'smaller' factors if they exist) is used with parameters that depend on the size of the remaining (composite) factors. After that, if a composite factor of at least 25 digits remains, MPQS is used; it is the best method available for factoring integers of more than about 40 decimal digits especially for products of two primes of roughly equal size. If the remaining composite is smaller than 25 digits, ECM is again invoked, now in an indefinite loop until a factor is found. The latter will also occur if the user, via a flag `MPQSLimit` indicates that MPQS should not be applied to numbers of the given size, and provided the user has not limited the number of ECM trials by setting the `ECMLimit`. Thus, unless both `MPQSLimit` and `ECMLimit` are set as optional parameters by the users, the algorithm will continue until the complete factorization has been completed. Besides the limiting parameters just mentioned it also possible to avoid the use of primality proofs and receive probable primes, with a flag similar to that used on `IsPrime`; see the previous section. A verbose flag can be set to obtain informative printing on progress in the various stages of factorization. Specific flags for ECM and MPQS may be used as well; they are described in the next subsection. Since V2.27, parallel versions of the ECM and MPQS algorithms are also available and are selected by using the procedures `SetNthreads` and `StartWorkers(H, T): MonStgElt, RngIntElt`. See Subsection Integer Factorisation for more details.

`SetVerbose("Factorization", v): MonStgElt, RngIntElt`

(Procedure.) Set the verbose printing level for all of the factorization algorithms to be v. Currently the legal values for v are `true`, `false`, 0, or 1 (`false` is the same as 0, and `true` is the same as 1). If the level is 1, information is printed at each stage of the algorithm as a number is factored.

`Factorization(n): RngIntElt -> RngIntEltFact, RngIntElt, SeqEnum`

`Factorisation(n): RngIntElt -> RngIntEltFact, RngIntElt, SeqEnum`

`Factorization(n: parameters): RngIntElt -> RngIntEltFact, RngIntElt, SeqEnum`

`Factorisation(n: parameters): RngIntElt -> RngIntEltFact, RngIntElt, SeqEnum`

```
Proof             : BoolElt                    Default: true
Bases             : RngIntElt                  Default: 20
TrialDivisionLimit: RngIntElt                  Default: 10000
SQUFOFLimit       : RngIntElt                  Default: 24
PollardRhoLimit   : RngIntElt                  Default: 8191
ECMLimit          : RngIntElt                  Default:
MPQSLimit         : RngIntElt                  Default: Infinity()
```

A combination of algorithms (Cunningham, trial division, SQUFOF, Pollard ρ, ECM, and MPQS) is used to attempt to find the complete factorization of $|n|$, where n is a non-zero integer. A factorization sequence is returned, representing the completely factored part of $|n|$ (which is usually all of $|n|$). The second return value is 1 or -1, reflecting the sign of n. If the factorization could not be completed, a third sequence is returned, containing composite factors that could not be decomposed with the given values of the parameters; this can only happen if both `ECMLimit` and `MPQSLimit` have been set. (Note that the third variable will remain unassigned if the full factorization is found.)

When a very large prime (more than 200 decimal digits say), appears in the factorization, proving its primality may dominate the running time.

There are 6 optional parameters.

The parameter `Proof` (`Proof := true` by default) can be set to false to indicate that the first sequence may contain probable primes (see also the previous section), in which case the parameter `Bases` indicates the number of tests used by Miller-Rabin (`Bases := 20` by default).

The parameter `TrialDivisionLimit` can be used to specify an upper bound for the primes used in the trial division stage (default `TrialDivisionLimit := 10000`).

The parameter SQUFOFLimit can be used specify the maximum number of decimal digits for an integer to which SQUFOF should be applied; this is Shank's square form factorization method (default SQUFOFLimit := 24).

The parameter PollardRhoLimit can be used to specify an upper bound on the number of iterations in the ρ method (default PollardRhoLimit := 8191).

This optional parameter can be used to limit the number of curves used by the ECM part of the factorization attempt. Setting ECMLimit := 0 prevents the use of ECM. The default value depends on the size of the input, and ranges from 2 for n with less than 37 digits to around 500 for n with 80 digits. The smoothness is incremented in each step to grow by default from 500 to 600 (for 37 digits and less), and from 500 to about 10000 for n having 80 digits. For the indefinite case of ECM (which applies when MPQS is disallowed) the initial smoothness is 500, the number of curves is infinite and the smoothness is incremented by 100 in each step.

The parameter MPQSLimit can be used specify the maximum number of decimal digits for an integer to which MPQS should still be applied; MPQS will not be invoked on integers having less than (or sometimes equal) 25 decimal digits. Setting the parameter to anything less than 25 will therefore prevent MPQS from being used. Unless ECMLimit has been set, this will imply that ECM will be applied until the full factorization has been obtained.

Note that progress can be monitored by use of SetVerbose("Factorization", 1).

19.10.2 Storing Potential Factors

Since V2.14 (October 2007), Magma now internally stores a list of factors found by the ECM and MPQS algorithms. Subsequently, when either of those algorithms are to be invoked by the Factorization function, the integers in the list are first tried to see whether factors can be easily found. One may also give prime factors to Magma to store in this list via the following procedure.

StoreFactor(n): RngIntElt

StoreFactor(S): [RngIntElt]

(Procedure.) Store the single integer n or the integers in the set/sequence S in the list of factors to be tried by the Factorization function. Each integer must be a positive *prime*.

GetStoredFactors() -> [RngIntElt]

Return a sequence containing the currently stored integers.

ClearStoredFactors()

Clear the list of stored integer factors.

19.10.3 Specific Factorization Algorithms

In this subsection we discuss how various factorization algorithms can be accessed individually. Generally these function should not be used for ordinary factorization (for that use Factorization discussed in the previous subsection), but they can be used for experimentation, or to build a personal factorization function with control over each of the methods used. On some functions a little preprocessing is done to ensure that the argument is composite, that powers of 2 (and sometimes 3) are taken out and that the integer to be factored is not the power of an integer. For each of these functions the Proof (default true) and Bases parameters can be used to indicate that primality of prime factors need not be rigorously proved, and how many bases should be used in the compositeness test, as discussed in the subsection on IsPrime.

SetVerbose("Cunningham", b): MonStgElt, BoolElt

SetVerbose("ECM", b): MonStgElt, Elt

SetVerbose("MPQS", b): MonStgElt, Elt

Using this procedure to set either of the verbose flags "Cunningham", "ECM", or "MPQS", (which are false by default) enables the user to obtain progress information on attempts to factor integers using the 'Cunningham' method, ECM, or MPQS.

Cunningham(b, k, c): RngIntElt, RngIntElt, RngIntElt -> SeqEnum

This function attempts to factor $n = b^k + c$, where $c \in \{\pm 1\}$ and b and k are not too big. This function uses R. Brent's factor algorithm [BtR92], which employs a combination of table-lookups and attempts at 'algebraic' factorization (Aurifeuillian techniques). An error results if the tables, containing most of the known factors for numbers of this form (including the 'Cunningham tables'), cannot be located by the system. The function will always return the complete prime factorization (in the form of a factorization sequence) of the number n (but it may take very long before it completes); it should be pointed out, however, that the primes appearing in the factorization are only *probable primes* and a rigorous primality prover has not been applied.

AssertAttribute(RngInt, "CunninghamStorageLimit", l): Cat, MonStgElt, RngIntElt

This attribute is used to change the number of Cunningham factorizations which are stored in Magma. Normally, Magma stores a certain number of factorizations computed by the Cunningham intrinsic function so that commonly needed factorizations can be recalled quickly. When the stored list fills up, the factorization least recently accessed is removed from the list. Setting this attribute to zero ensures that no storage is done. The default value is 20.

TrialDivision(n): RngIntElt -> RngIntEltFact, RngIntElt

TrialDivision(n, B): RngIntElt, RngIntElt -> RngIntEltFact, RngIntElt

The integer $n \neq 0$ is subjected to trial division by primes up to a certain bound B (the sign of n is ignored). If only the argument n is given, B is taken to be 10000. Since V2.29-9, the function returns the factorization sequence of the part of $|n|$ which is composed of the primes up to B, together with the residue divisor of $|n|$ which remains; the residue is 1 if and only if $|n|$ is completely factored by the trial division. The bound B is respected exactly, so no prime greater than B ever occurs in the factorization sequence. (In versions before V2.29-9 the primes 2, 3, 5, and 7 were always divided out, whatever B was.) No primality test is performed on the residue: if it is desired to know whether the residue is prime, IsPrime should be applied to it afterwards.

For backwards compatibility (with versions before V2.29-9), the parameters Proof and Bases are still accepted but are now ignored (in earlier versions they controlled a primality test which was applied to the residue).

PollardRho(n): RngIntElt -> RngIntEltFact, [RngIntElt]

PollardRho(n, c, s, k): RngIntElt, RngIntElt, RngIntElt, RngIntElt -> RngIntEltFact, [RngIntElt]

Proof: BoolElt	Default: true
Bases: RngIntElt	Default: 20

The ρ-method of Pollard is invoked by this function to find the factorization of an integer $n > 1$. For this method a quadratic function $x^2 + c$ is iterated k times, with starting value $x = s$. If only n is used as argument to the function, the default values $c = 1$, $s = 1$, and $k = 8191$ are selected. A speed-up to the original algorithm, due to R. P. Brent [Bre80], is implemented. The function returns two values: a factorization sequence and a sequence containing unfactored composite factors.

pMinus1(n, B1): RngIntElt, RngIntElt -> RngIntElt

x0: RngIntElt	Default:
B2: RngIntElt	Default:
k : RngIntElt	Default:

Given an integer $n > 1$, an attempt to find a factor is made using Paul Zimmermann's GMP-ECM implementation of Pollard's $p-1$ method. If a factor f with $1 < f < n$ is found, then f is returned; otherwise 0 is returned.

The Step 1 bound B_1 is given as the second argument B1. By default, the Step 2 bound B_2 is optimally chosen, but may be given with the parameter B2 instead. By default, an optimal number of blocks is chosen for Step 2, but this may be overridden via the parameter k (see the function ECM). The base x_0 is chosen randomly by default, but may instead be supplied via the parameter x0.

This method will return a prime factor p of n if $p-1$ has all its prime factors less than or equal to the Step 1 bound B_1, except for one factor which may be less than or equal to the Step 2 bound B_2.

`pPlus1(n, B1): RngIntElt, RngIntElt -> RngIntElt`

x0: RngIntElt	Default:
B2: RngIntElt	Default:
k : RngIntElt	Default:

Given an integer $n > 1$, an attempt to find a factor is made using Paul Zimmermann's GMP-ECM implementation of Williams' $p+1$ method. If a factor f with $1 < f < n$ is found, then f is returned; otherwise 0 is returned.

The Step 1 bound B_1 is given as the second argument `B1`. By default, the Step 2 bound B_2 is optimally chosen, but may be given with the parameter `B2` instead. By default, an optimal number of blocks is chosen for Step 2, but this may be overridden via the parameter `k` (see the function `ECM`). The base x_0 is chosen randomly by default, but may instead be supplied via the parameter `x0`.

This method may return a prime factor p of n if $p+1$ has all its prime factors less than or equal to the Step 1 bound B_1, except for one factor which may be less than or equal to the Step 2 bound B_2. A base x_0 is used, and not all bases will succeed: only half of the bases work (namely those where the Jacobi symbol of $x_0^2 - 4$ and p is -1.) Unfortunately, since p is usually not known in advance, there is no way to ensure that this holds. However, if the base is chosen randomly, there is a probability of about 1/2 that it will give a Jacobi symbol of -1 (so that the factor p would be found assuming that $p+1$ is smooth enough). A rule of thumb is to run `pPlus1` three times with different random bases.

`SQUFOF(n): RngIntElt -> RngIntEltFact, [ RngIntElt ]`

`SQUFOF(n, k): RngIntElt, RngIntElt -> RngIntEltFact, [ RngIntElt ]`

Proof: BoolElt	Default: true
Bases: RngIntElt	Default: 20

This is a fast implementation of Shanks's square form factorization method that will only work for integers $n > 1$ less than 2^{2b-2}, where b is the number of bits in a long (which is either 32 or 64). The argument k may be used to specify the maximum number of iterations used to find the square; by default it is 200000. The expected number of iterations is $O(N^{1/4})$.

`ECM(n, B1): RngIntElt, RngIntElt -> RngIntElt, RngIntElt`

Sigma: RngIntElt	Default:
x0 : RngIntElt	Default:
B2 : RngIntElt	Default:
k : RngIntElt	Default: 2

Given an integer $n > 1$, an attempt is made to find a factor using the GMP-ECM implementation of the Elliptic Curve Method (ECM). If a factor f with $1 < f < n$ is found, then f is returned together with the corresponding successful σ seed; otherwise 0 is returned.

The Step 1 bound B_1 is given as the second argument `B1`. By default, the Step 2 bound B_2 is optimally chosen, but may be given with the parameter `B2` instead.

The elliptic curve used is defined by Suyama's parametrization and is determined by a parameter σ. By default, σ is chosen randomly with $0 < \sigma < 2^{32}$, but an alternative positive integer may be supplied instead via the parameter `Sigma`. Let $u = \sigma^2 - 5$, $v = 4\sigma$, and $a = (v-u)^3(3u+v)/(4u^3v) - 2$. The starting point used is $(x_0 : 1)$, where by default $x_0 = u^3/v^3$, but x_0 may instead be supplied via the parameter `x0`. Finally, the curve used is $by^2 = x^3 + ax^2 + x$, where $b = x_0^3 + ax_0^2 + x_0$.

Step 1 uses very little memory, but Step 2 may use a large amount of memory, especially for large B_2, since its efficient algorithms use some large tables. To reduce the memory usage of Step 2, one may increase the parameter k, which controls the number of "blocks" used. Multiplying the default value of k by 4 will decrease the memory usage by a factor of 2. For example, with $B_2 = 10^{10}$ and a 155-digit number n, Step 2 requires about 96MB with the default $k = 2$, but only 42MB with $k = 8$. Increasing k does, however, slightly increase the time required for Step 2.

ECMSteps(n, L, U): RngIntElt, RngIntElt, RngIntElt -> RngIntElt, RngIntElt

Given an integer $n > 1$, an attempt to find a factor of n is made by repeated calls to ECM. The initial B_1 bound is taken to be L, and subsequently B_1 is replaced with $B_1 + \lfloor\sqrt{B_1}\rfloor$ at each step. If a factor is found at any point, then this is returned with the corresponding successful σ seed; otherwise, if B_1 becomes greater than the upper bound U, then 0 is returned.

MPQS(n): RngIntElt -> RngIntEltFact, [RngIntElt]

Proof: BoolElt	Default: true
Bases: RngIntElt	Default: 20

This function can be used to drive Arjen Lenstra's implementation of the multiple polynomial quadratic sieve MPQS. Given an integer $n > 5 \cdot 10^{24}$ an attempt is made to find the prime factorization of n using MPQS. The function returns two values: a factorization sequence and a sequence containing unfactored composite factors.

19.10.4 Factorization Related Functions

ECMOrder(p, s): RngIntElt, RngIntElt -> RngIntElt

ECMFactoredOrder(p, s): RngIntElt, RngIntElt -> RngIntElt

Suppose p is a prime factor found by the ECM algorithm and such that the σ value determining the successful curve was s. These functions compute the order of the corresponding elliptic curve. The first function returns the order as an integer, while the second function returns the factorization of the order. In general, this order will have been smooth with respect to the relevant bounds for the ECM algorithm to have worked, and these functions allow one to examine how small the prime divisors of the curve order really are.

PrimeBasis(n): RngIntElt -> [RngIntElt]

PrimeDivisors(n): RngIntElt -> [RngIntElt]

A sequence containing the distinct prime divisors of the positive integer $|n|$, given in increasing order.

Divisors(n): RngIntElt -> [RngIntElt]

Divisors(f): RngIntEltFact -> [RngIntElt]

Returns a sequence containing all divisors of the positive integer, including 1 and the integer itself, given in increasing order. The argument given must be either the integer n itself, or a factorization sequence f representing it.

CoprimeBasis(S): [RngIntElt] -> [RngIntElt]

CoprimeBasis(S): { RngIntElt } -> [RngIntElt]

Given a set or sequence S of integers, return a coprime basis of S in the form of a factorization sequence Q whose integer value is the same as the product of the elements of S but Q has coprime bases (i.e., the first components of tuples from Q are coprime).

Example: Perfect Numbers (ex-07a3e8)

In this example we use the Divisors function together with the &+ reduction of sequences to find the first few perfect numbers, that is, numbers n such that the sum of the divisors less than n equals n.

```
> {  x : x in [2..1000] | &+Divisors(x) eq 2*x };
{ 6, 28, 496 }
```

(continues on next page)

(continued from previous page)

```
> f := Factorization(496);
> f;
[ <2, 4>, <31, 1> ]
> Divisors(f);
[ 1, 2, 4, 8, 16, 31, 62, 124, 248, 496 ]
```

PartialFactorization(S): [RngIntElt] -> [RngIntEltFact]

Given a sequence of non-zero integers S, return, for each integer $S[i]$, two factorization lists F_i and G_i, such that $S[i] =$ Facint$(F_i)*$Facint(G_i). All the divisors in F_i are square factors, and, for any i and j, the divisors in G_i and G_j are either equal or are pairwise coprime. In other terms, PartialFactorization(S) provides a partial decomposition of the integers in S in square and coprime factors. The interesting fact is that this factorization uses only gcd and exact integer division. This algorithm is due to J.E. Cremona.

Example: Partial Factorization (ex-15e5a0)

A partial factorization is shown.

```
> PartialFactorization([1380, 675, 3408, 654]);
[
  [
    [ <2, 2> ],
    [ <115, 1>, <3, 1> ]
  ],
  [
    [ <5, 2>, <3, 2> ],
    [ <3, 1> ]
  ],
  [
    [ <2, 4> ],
    [ <71, 1>, <3, 1> ]
  ],
  [
    [],
    [ <218, 1>, <3, 1> ]
  ]
]
```

19.11 Factorization Sequences

The factorization of integers results in a factorization sequence, consisting of a sequence of pairs of prime and exponent. It is sometimes convenient to perform operations on such sequences without converting back to the integers they represent — it would, for example, be very inefficient to factor the product of two integers that have both been factored already. In this section we briefly list the operations that are allowed on such factorization sequences — note that these factorization sequences now have their own special type: RngIntEltFact. Conversion functions are supplied as well.

19.11.1 Creation and Conversion

Factorization sequence usually arise as the result of the Factorization of an integer, possibly via functions like FactoredOrder. The functions below allow conversion from and to ordinary sequences, and the inverse operation to factorization, creating an integer from a factorization.

Facint(f): RngIntEltFact -> RngIntElt

FactorizationToInteger(f): RngIntEltFact -> RngIntElt

Create the integer corresponding to the factorization sequence f.

SeqFact(s): SeqEnum -> RngIntEltFact

SequenceToFactorization(s): SeqEnum -> RngIntEltFact

Given a sequence of tuples, each consisting of pairs of prime integers and positive integer exponents, create the corresponding factorization sequence. The pairs must be ordered with strictly increasing primes as first components.

Eltseq(f): RngIntEltFact -> SeqEnum

ElementToSequence(f): RngIntEltFact -> SeqEnum

Given a factorization sequence f, create the enumerated sequence containing the same pairs of primes and exponents.

19.11.2 Arithmetic

The difference of two factorization sequences is only permitted when the first integer represented is greater than the second integer represented. An error results from division when the quotient does not correspond to an integer.

s + t: RngIntEltFact, RngIntEltFact -> RngIntEltFact

s - t: RngIntEltFact, RngIntEltFact -> RngIntEltFact

s * t: RngIntEltFact, RngIntEltFact -> RngIntEltFact

s / t: RngIntEltFact, RngIntEltFact -> RngIntEltFact

s ^ k: RngIntEltFact, RngIntElt -> RngIntEltFact

19.11.3 Divisors

The functions listed below can be applied to factorization sequences; their behaviour will be clear, and all of them are documented elsewhere when the argument is the corresponding positive integer.

Lcm(s, t): RngIntEltFact, RngIntEltFact -> RngIntEltFact

Gcd(s, t): RngIntEltFact, RngIntEltFact -> RngIntEltFact

SquarefreeFactorization(f): RngIntEltFact -> RngIntEltFact, RngIntEltFact

MoebiusMu(f): RngIntEltFact -> RngIntElt

Divisors(f): RngIntEltFact -> SeqEnum

PrimeDivisors(f): RngIntEltFact -> SeqEnum

NumberOfDivisors(f): RngIntEltFact -> RngIntElt

SumOfDivisors(f): RngIntEltFact -> RngIntElt

19.11.4 Predicates

All predicates listed below are applicable both to factorization sequences and to the positive integers these represent, and have been documented for integer arguments elsewhere.

IsOne(s): RngIntEltFact -> BoolElt

IsOdd(s): RngIntEltFact -> BoolElt

IsEven(s): RngIntEltFact -> BoolElt

IsUnit(s): RngIntEltFact -> BoolElt

IsPrime(s): RngIntEltFact -> BoolElt

`IsPrimePower(s): RngIntEltFact -> BoolElt`

`IsSquare(s): RngIntEltFact -> BoolElt`

`IsSquarefree(s): RngIntEltFact -> BoolElt`

19.12 Modular Arithmetic

In this section we describe some functions that make it possible to perform modular arithmetic without conversions to residue class rings.

19.12.1 Arithmetic Operations

`Modexp(n, k, m): RngIntElt, RngIntElt, RngIntElt -> RngIntElt`

The modular power $n^k \bmod m$, where n is an integer, k is an integer and m is an integer greater than one. If k is negative, n must have an inverse i modulo m, and the result is then $i^{-k} \bmod m$. The result is always an integer r with $0 \leq r < m$.

`n mod m: RngIntElt, RngIntElt -> RngIntElt`

Remainder upon dividing the integer n by the integer m. The result always has the same sign as m. An error results if m is zero.

`Modinv(n, m): RngIntElt, RngIntElt -> RngIntElt`

`InverseMod(n, m): RngIntElt, RngIntElt -> RngIntElt`

Given an integer n and a positive integer m, such that n and m are coprime, return an inverse u of n modulo m, that is, return an integer $1 \leq u < m$ such that $u \cdot n \equiv 1 \bmod m$.

`Modsqrt(n, m): RngIntElt, RngIntElt -> BoolElt, RngIntElt`

Given an integer n and an integer $m \geq 2$, this function returns an integer b such that $0 \leq b < m$ and $b^2 \equiv n \bmod m$ if such b exists; an error results if no such root exists.

`Modorder(n, m): RngIntElt, RngIntElt -> RngIntElt`

For integers n and m, $m > 1$, the function returns the least integer $k \geq 1$ such that $n^k \equiv 1 \bmod m$, or zero if $\gcd(n, m) \neq 1$.

`IsPrimitive(n, m): RngIntElt, RngIntElt -> BoolElt`

Returns `true` if n is a primitive root for m, `false` otherwise ($0 < n < m$).

`PrimitiveRoot(m): RngIntElt -> RngIntElt`

Given an integer $m > 1$, this function returns an integer value defined as follows: If $\mathbb{Z}/m\mathbb{Z}$ has a primitive root and the function is successful in finding it, the root a is returned. If $\mathbb{Z}/m\mathbb{Z}$ has a primitive root but the algorithm does not succeed in finding it, or $\mathbb{Z}/m\mathbb{Z}$ does not possess a primitive root, then zero is returned.

19.12.2 The Solution of Modular Equations

The functions described here can be used if an occasional modular operation is required; the results are integers again. For more extensive modular arithmetic it is preferable to convert to residue class ring arithmetic. See section *Residue Class Rings* for details.

`Solution(a, b, m): RngIntElt, RngIntElt, RngIntElt -> RngIntElt, RngIntElt`

If a solution exists to the linear congruence $ax \equiv b \bmod m$, then returns $x0, k$ such that $x = x0 + i * k$ represents the complete set of solutions, where i can be any integer. Otherwise, returns -1.

`ChineseRemainderTheorem(X, N): [RngIntElt], [RngIntElt] -> RngIntElt`

`CRT(X, N): [RngIntElt], [RngIntElt] -> RngIntElt`

Apply the Chinese Remainder Theorem to the integer sequences X and N. The sequences must have the same length, k say. The function returns the unique integer x in the range $0 \leq x < LCM(N[1] \cdot \ldots \cdot N[k])$ such that $x \equiv X[i] \bmod N[i]$. The elements of N must all be positive integers greater than one. If there is no solution, then -1 is returned.

Solution(A, B, N): [RngIntElt], [RngIntElt], [RngIntElt] -> RngIntElt

Return a solution x to the system of simultaneous linear congruences defined by the integer sequences A, B, and N. Each of these sequences must have the same number of terms, k say. The elements of N must all be positive integers greater than one. The i-th congruence is $A[i] \cdot x \equiv B[i] \bmod N[i]$. The solution x will satisfy $0 \leq x < LCM(N[1] \cdot \ldots \cdot N[k])$. If no solution exists, -1 is returned.

NormEquation(d, m): RngIntElt, RngIntElt -> BoolElt, RngIntElt, RngIntElt

NormEquation(d, m: parameters)): RngIntElt, RngIntElt -> BoolElt, RngIntElt, RngIntElt

Factorization: [<RngIntElt, RngIntElt>]	Default: [ ]

Given a positive integer d and a non-negative integer m, return **true** and two non-negative integers x and y, such that $x^2 + y^2 d = m$, if such a solution exists. If such a solution does not exists only the value **false** is returned. If the factorization of m is known, it may be supplied as the value of the parameter **Factorization** to speed up the computation.

Example: Norm Equation (ex-a39523)

```
> d := 957440000095744000002277749760;
> m := 5102197760510219776012138128480644;
> time NormEquation(d, m);
true 98 73
Time: 2.990
> time f := Factorization(m);
Time: 4.670
> f;
[ <2, 2>, <19, 1>, <671341810593449970527912911642l9, 1> ]
> time NormEquation(d, m: Factorization := f);
true 98 73
Time: 0.420
```

19.13 Infinities

Occasionally it is convenient to work with infinite quantities (for example, when working with valuations or cardinalities). Magma provides two such objects, the positive and negative infinities. This section describes the Magma facilities for dealing with such objects.

The infinities are compatible with certain finite quantities: integers, rationals, and real numbers. In contexts where a common universe is needed to contain both finite and infinite quantities (for example, if creating a sequence of valuations) the extended reals (type **ExtRe**) are used. The extended reals are a coproduct-like object that can contain both infinities and compatible finite objects. When viewed as members of the extended reals, the elements are of type **ExtReElt**.

19.13.1 Creation

Certain system intrinsics such as **Valuation** which normally return an integer may return an infinite object for appropriate exceptional cases. Two special intrinsics are also provided to create infinite objects.

Infinity() -> Infty

The positive infinity object.

MinusInfinity() -> Infty

The negative infinity object.

19.13.2 Arithmetic

Only basic arithmetic operations are provided for infinite objects. The operations described below may freely mix infinite and finite quantities, but note that certain forms (such as $\infty - \infty$ or $\infty * 0$) are not well defined and will cause an error.

```
- x: Infty -> Infty

x + y: Infty, Infty -> Infty

x + y: Infty, RngIntElt -> Infty

x + y: RngIntElt, Infty -> Infty

x - y: Infty, Infty -> Infty

x - y: Infty, RngIntElt -> Infty

x - y: RngIntElt, Infty -> Infty

x * y: Infty, Infty -> Infty

x * y: Infty, RngIntElt -> Infty

x * y: RngIntElt, Infty -> Infty

x / y: RngIntElt, Infty -> RngIntElt

x / y: Infty, RngIntElt -> Infty

x ^ n: Infty, Infty -> Infty

x ^ n: Infty, RngIntElt -> Infty
```

19.13.3 Comparison

Infinite objects may be compared with themselves and finite quantities.

```
x eq y: Infty, Infty -> BoolElt

x eq y: Infty, RngIntElt -> BoolElt

x eq y: RngIntElt, Infty -> BoolElt

x ne y: Infty, Infty -> BoolElt

x ne y: Infty, RngIntElt -> BoolElt

x ne y: RngIntElt, Infty -> BoolElt

x lt y: Infty, Infty -> BoolElt

x lt y: Infty, RngIntElt -> BoolElt

x lt y: RngIntElt, Infty -> BoolElt

x le y: Infty, Infty -> BoolElt

x le y: Infty, RngIntElt -> BoolElt

x le y: RngIntElt, Infty -> BoolElt

x gt y: Infty, Infty -> BoolElt

x gt y: Infty, RngIntElt -> BoolElt
```

`x gt y: RngIntElt, Infty -> BoolElt`

`x ge y: Infty, Infty -> BoolElt`

`x ge y: Infty, RngIntElt -> BoolElt`

`x ge y: RngIntElt, Infty -> BoolElt`

`Maximum(x, y): Infty, Infty -> .`

`Maximum(x, y): Infty, RngIntElt -> .`

`Maximum(x, y): RngIntElt, Infty -> .`

`Minimum(x, y): Infty, Infty -> .`

`Minimum(x, y): Infty, RngIntElt -> .`

`Minimum(x, y): RngIntElt, Infty -> .`

19.13.4 Miscellaneous

`Sign(x): Infty -> RngIntElt`

Returns 1 if x is the positive infinite object, -1 if x is the negative infinite object.

`Abs(x): Infty -> Infty`

`AbsoluteValue(x): Infty -> Infty`

Returns the positive infinite object.

`Round(x): Infty -> Infty`

`Floor(x): Infty -> Infty`

`Ceiling(x): Infty -> Infty`

Returns the infinite object x again; these functions are for convenience when dealing with objects which could be either finite numeric types or infinite objects.

`IsFinite(x): Infty -> BoolElt`

`IsFinite(x): RngIntElt -> BoolElt`

Returns `true` if x is finite, otherwise `false`. This is more convenient than checking the type of x.

19.14 Advanced Factorization Techniques: The Number Field Sieve

Magma provides an experimental implementation of the fastest general purpose factoring algorithm known: the Number Field Sieve (NFS). The implementation may be used both as a General Number Field Sieve and a Special Number Field Sieve – the only difference is in the selection of a suitable polynomial.

19.14.1 The Magma Number Field Sieve Implementation

In order to make use of the Magma NFS, the user should have some knowledge of the algorithm. The Magma NFS implementation also requires a significant amount of memory and disk space to be available for the duration of the factorization. For example, factorization of an 80-digit number may require at least 64 megabytes of RAM and half a gigabyte of disk space.

Magma's NFS implementation uses one linear polynomial (the "rational side") and one polynomial of higher degree (the "algebraic side"). At the time of writing this is not a major restriction, since the best methods for selecting polynomials for factorization of numbers of more than 100 digits involve one linear and one non-linear polynomial. Magma provides a number of functions to assist in choosing a good algebraic-side polynomial for the factorization of a particular number, following the ideas of Montgomery and Murphy in [Mur99].

Magma provides two methods for using the NFS implementation. The first is the one-step function NFS, which provides a naive NFS factorization attempt using default algorithm parameters.

The second, more powerful method is to work with an NFS process object, splitting the algorithm into four stages: Sieving, Auxiliary data, Linear algebra, and Final factorization. This approach allows greater control over the algorithm, as the user may supply their own algorithm parameter values. It also allows the user to distribute the computationally intensive sieving and final factorization stages over several machines or processors.

Some functions are included to allow Magma users to co-operate in factorization attempts using CWI tools.

A verbose flag may be set to obtain informative printing on progress in the various stages of the NFS algorithm.

`SetVerbose("NFS", v): MonStgElt, RngIntElt`

Set the verbose printing level for the NFS algorithms to the integer v. Currently the legal values for v are 0, 1, 2, and 3. If the level is 0, no verbose output is produced. If the level is 1, NFS will produce basic information about its progress, and will also print information on NFS algorithm parameters. If the level is 2, NFS will provide more detailed information about progress and parameters. If the level is 3, NFS will print out extremely detailed information about progress and data. This level will only be useful for experts and developers.

19.14.2 Naive NFS

Magma's Number Field Sieve implementation provides a one-step black-box function NFS. Here, the user provides the integer n to be factored, a homogeneous bivariate integer polynomial F, and integers $m1$ and $m2$ such that $F(m1, m2) \equiv 0 \bmod n$. Magma will attempt to factor n using F, $m1$, and $m2$, automatically selecting the other parameters (see below) for the algorithm.

The automatically chosen parameters are NOT optimal in general, and therefore no conclusions should be drawn about the speed of the implementation or the algorithm itself based on the use of this function.

For example, note that the default algebraic factor base size of NFS is chosen to be rather large to decrease the likelihood of running out of useful relations. This slows the algorithm considerably, since it increases the size of the matrix to be reduced – but it also means that the algorithm should succeed in finding a factor unless one chooses a really bad polynomial.

`NumberFieldSieve(n, F, m1, m2): RngIntElt, RngMPolElt, RngIntElt, RngIntElt -> RngIntElt`

`NFS(n, F, m1, m2): RngIntElt, RngMPolElt, RngIntElt, RngIntElt -> RngIntElt`

Performs the factorization of an integer n using the Number Field Sieve with algebraic polynomial F, where the integers $m1$ and $m2$ satisfy $F(m1, m2) \equiv 0 \bmod n$. Returns a nontrivial factor of n if one is found, or 0 otherwise.

19.14.3 Factoring with NFS Processes

An NFS Process (an object of category NFSProc) encapsulates the data of a Magma NFS factorization. It contains the number n to be factored, the algebraic polynomial F, and the integers m_1 and m_2. It also provides access to a number of NFS algorithm parameters (such as approximate factor base sizes). These parameters are attributes of the NFS process. If any of the parameters are not set, sensible (but not necessarily optimal) defaults will be provided by Magma.

The NFS algorithm is divided into four stages:

1. Sieving
2. Auxiliary data gathering
3. Linear algebra
4. Factorization

The stages are described in detail below.

After creating an NFS process for the factorization attempt, the user should proceed through each of the four stages in the above order.

`NFSProcess(n, F, m1, m2): RngIntElt, RngMPolElt, RngIntElt, RngIntElt -> NFSProc`

Given a (composite) integer n, a bivariate homogeneous integer polynomial F, and nonzero integers $m1$ and $m2$ such that $F(m1, m2) \equiv 0 \bmod n$, this function creates an NFS process object for an NFS factorization of n.

Example: NFS Process Attributes (ex-53bcfa)

The attributes associated with an NFS process are:

```
> ListAttributes(NFSProc);
AlgebraicError              OutputFilename
AlgebraicFBBound            RationalError
AlgebraicLargePrimeBound    RationalFBBound
CacheSize                   RationalLargePrimeBound
F                           m1
Firstb                      m2
Lastb                       n
Maximuma
```

`OutputFilename` is the base name for NFS-generated data files. These files (and their actual names) are discussed below.

`AlgebraicFBBound` is the upper bound for smooth primes in the algebraic-side factor base, and `RationalFBBound`, the upper bound for smooth primes in rational-side factor base.

`Maximuma` bounds the sieve interval for a: NFS will sieve for relations with $|a| \leq$ `Maximuma`.

`Firstb` is the first value of b to sieve on, and `Lastb` is the last.

`AlgebraicLargePrimeBound` gives the upper bound for "large" (non-smooth) primes in the algebraic-side factor base. Similarly, `RationalLargePrimeBound` is the upper bound for the rational side.

`AlgebraicError` defines an "error" tolerance for logarithm arithmetic on algebraic side. Similarly, `RationalError` defines an "error" tolerance for the rational side.

`CacheSize` is a flag reflecting the computer cache memory size, for optimisation.

Attribute Selection

As a guideline for the selection of attributes, we include here a few examples of attributes that we have determined to be good for the Magma NFS implementation.

Example: NFS Attributes for a 70-Digit Number (ex-25ef84)

Sample attributes for a 70-digit number:

```
> n := 523586968023336629536690451072545805304311124103567889793380223506092 7;
> R<X,Y> := PolynomialRing(Integers( ), 2);
> F := 2379600*X^4 - 12052850016*X^3*Y - 13804671642407*X^2*Y^2 +
>      11449640164912254*X*Y^3 + 7965530070546332840*Y^4 ;
> m1 := 6848906180202117;
> m2 := 1;
> P := NFSProcess(n,F,m1,m2);
> P`AlgebraicFBBound := 8*10^5;
> P`RationalFBBound := 6*10^5;
> P`OutputFilename := "/tmp/nfs_70_digit";
> P`Maximuma := 4194280;
> P`AlgebraicError := 16;
> P`RationalError := 14;
```

Example: NFS Attributes for an 80-Digit Number (ex-9e7234)

Sample attributes for an 80-digit number:

```
> n := 18718318663576864934511227229510402220632793503837386502539069334890 72\\
> 2483083589;
> P<X,Y> := PolynomialRing(Integers(),2);
> F := 15901200*X^4 + 9933631795*X^3*Y - 112425819157429*X^2*Y^2 -
>     231659214929438137*X*Y^3 - 73799500175565303965*Y^4;
> m1 := 1041619817688573426;
> m2 := 1;
> P := NFSProcess(n, F, m1, m2);
> P`AlgebraicFBBound := 8*10^5;
> P`RationalFBBound := 6*10^5;
> P`OutputFilename := "/tmp/nfs_80_dgit";
> P`Maximuma := 10485760;
> P`AlgebraicError := 16;
> P`RationalError := 14;
```

Example: NFS Attributes for an 87-Digit Number (ex-7d7073)

Sample attributes for an 87-digit number:

```
> n := 12118618732463427472219179104631767765107839384219612469780841876821498\\
> 2402918637227743;
> P<X,Y> := PolynomialRing(Integers(),2);
> F := 190512000*X^4 - 450872401242*X^3*Y +
>     1869594915648551*X^2*Y^2 + 2568544235742498*X*Y^3 -
>     93229655834198010101 04*Y^4;
> m1 := 28241170741195273211;
> m2 := 1;
> P := NFSProcess(n, F, m1, m2);
> P`AlgebraicFBBound := 16*10^5;
> P`RationalFBBound := 10^6;
> P`OutputFilename := "/tmp/nfs_87_digit";
> P`Maximuma := 2^24;
> P`AlgebraicError := 24;
> P`RationalError := 18;
```

The best choice for the factor base size depends on many variables, including the average log size and the Murphy α parameter (defined in [Mur99]) for the polynomial F. Our polynomials above are quite good: if the user does not know much about determining the quality of polynomials, then he or she should use much larger factor bases.

The Sieving Stage

Magma's NFS uses a "line-by-line" (or "classical") sieving algorithm. Future versions may include lattice sieving.

The line-by-line siever sieves values of $F(a, b)$ on the algebraic side and corresponding values $a \cdot m2 - b \cdot m1$ on the rational side. This is done by fixing a value of b (beginning with the parameter `Firstb`, if supplied), then sieving all values of a between $-a_0$ and a_0, where a_0 is approximately equal to the parameter `Maximuma` (some rounding off is done to make sure that the sieve interval length is divisible by a high power of 2). When this is completed b is incremented, and the next value of b is processed.

The sieving continues until either the maximum value of b (specified by the parameter `Lastb`) has been reached, or until enough relations are obtained to complete the factorization. If `Lastb` is not defined, the sieve simply continues until enough relations are found. The number of relations required may be determined by the function `NumberOfRelationsRequired`.

"Cycles" among partial relations are counted after every 256 iterations.

The sieve implementation uses (rounded natural) logarithms of primes to mark the sieve interval. Moreover, the implementation does not sieve with prime powers. Therefore, we must allow for some error in scanning the sieve arrays for useful relations; the acceptable sieve threshold errors for each side are defined by the `AlgebraicError` and `RationalError` parameters. If, in addition, the user wants to take advantage of large prime relations (recommended), then larger error terms should be used. The implementation will keep relations having up to 2 large primes on each side, but will only find such relations if the user selects large enough sieve threshold error bounds. The user should be cautious when sieving for (and subsequently using) relations with large primes, as they greatly increase overall disk space requirements. Some experimentation may be required in order to determine the best error bounds for speed or disk space optimization purposes.

The `CacheSize` parameter may be used to take advantage of the cache memory size of the computer: a value of 1 indicates a small cache size, 2 a medium cache size, and 3 a for large cache size.

`NumberOfRelationsRequired(P): NFSProc -> RngIntElt`

The minimum number of relations required for an NFS factor attempt with NFS process P.

`FindRelations(P): NFSProc -> RngIntElt`

Given an NFS process P for factoring an integer n, generates relations to factor n with the Number Field Sieve algorithm. Returns the number of full relations plus the number of cycles found.

The Auxiliary Data Stage

In this stage of the algorithm, "cycles" [LD95] are detected in the partial relations from the sieving stage, and quadratic characters are calculated for the relations. This greatly improves the efficiency of the NFS.

In a typical factorization, the user should call the procedures `CreateCycleFile` and `CreateCharacterFile` in succession.

`CreateCycleFile(P): NFSProc -> .`

Creates a file with all the cycle information that the NFS algorithm requires to complete the matrix reduction and final factorization stages for the NFS process P.

`CycleCount(P): NFSProc -> RngIntElt`

Returns the number of cycles in the partial relations of the NFS process P. This function is mainly intended for factoring with multiple processors.

`CycleCount(fn): MonStgElt -> RngIntElt`

Returns the number of cycles in the partial data file corresponding to the base file name fn. This function is mainly intended for factoring with multiple processors.

`CreateCharacterFile(P): NFSProc -> .`

Creates a file with the quadratic character data for the full relations and cycles in the NFS process P.

`CreateCharacterFile(P, cc): NFSProc, RngIntElt -> .`

Creates a file with the quadratic character data for the full relations and cycles in the NFS process P. There are cc sets of 32 quadratic character columns created.

Finding Dependencies: the Linear Algebra Stage

In this stage, the relations are collected together to form a matrix, and then block Lanczos reduction is applied to find linear dependencies among the relations. These dependencies become candidates for factorization.

`FindDependencies(P): NFSProc -> .`

Finds dependencies between relations in the NFS process P.

The Factorization Stage

In this stage, number field square roots are extracted and we attempt to factor the dependencies found in the linear algebra stage.

`Factor(P): NFSProc -> RngIntElt`

Try to factor with each dependency in the NFS process P until a proper factor is found. Returns the factor, or 0 if no factor is found.

`Factor(P, k): NFSProc, RngIntElt -> RngIntElt`

Attempt to factor with the k-th dependency in the NFS process P. Returns a proper factor if found, 0 otherwise.

19.14.4 Data Files

Many data files are ued for an NFS factorization. The user can control the names and location of the files by specifying the `OutputFilename` parameter; then all output files will have names beginning with the `OutputFilename` string, with a range of suffixes depending on their purpose. In general, all files are appended to rather than overwritten; so to avoid inconsistencies (and to save disk space) the user should call `RemoveFiles` after a successful factorization.

When distributing factorizations, or collecting results from sieving stages that have been broken up into several runs for some reason (for example, if a process has been interrupted), Magma provides the function `MergeFiles`. This takes a sequence of base filenames (which are treated as if they were the value for `OutputFilename`), and reads in the corresponding relation and partial relation files; it then combines the contents of these files, removing duplicates and corrupted lines of data, and places the results into new relation and partial relation files.

`RemoveFiles(P): NFSProc -> .`

Deletes any data files created by the NFS process P.

`MergeFiles(S, fn): [MonStgElt], MonStgElt -> RngIntElt, RngIntElt`

Merges the NFS relation files named in the sequence S (and their associated partial relation files) into a pair of new relation and partial relation files, while removing duplicate and corrupted lines of data; returns the number of relations and the number of partial relations in the new output files. The combined full relations are stored in a file named fn, and the partial relations in a file named fn_partials.

Magma Native NFS Data Files

Here we describe the files used in a typical Magma NFS factorization. These files all use formats peculiar to `Magma`'s NFS.

The first kind of file created by NFS stores the relations generated in the sieving stage by the `FindRelations` procedure. The name of the file is precisely the `OutputFilename` string.

NFS also stores partial relations generated in the sieving stage; these are stored in a file named `OutputFilename`_partials.

Whenever cycles [LD95] are counted (for example, in `CycleCount`, a file named `OutputFilename`_cycles is created to store them in. Some other files are also created and then deleted during the cycle counting process.

The quadratic characters calculated in `CreateCharacterFile` are stored in a file named `OutputFilename`_cc.

The linear algebra stage creates a file named `OutputFilename`_null_space, which lists relations making up null space vectors for the NFS matrix.

19.14.5 Distributing NFS Factorizations

Magma provides a number of tools for distributing the sieving and final factoring stages over a number of computers.

To distribute the sieving stage, each processor should get a unique range of b-values to sieve and unique data file names. During the sieving, the user must manually check when the combined data has enough relations to factor the number. To do this, the data files must first be merged using `MergeFiles`, and then the cycles can be counted with `CycleCount`. If the combined number of full relations plus the number of cycles exceeds the size of both factor bases combined, then the user can proceed to the other stages of the factorization attempt using the merged data file name.

To distribute the factorization stage, the user may choose a dependency for each process to factor, then call `Factor(P,k)` where P is the NFS process and k the number specifying the dependency to factor, with a different value of k for each process.

Example: Distributed NFS Factorization (ex-dc02d5)

Here we demonstrate a distributed NFS factorization (of a very small n) over two processes, A and B – which may be on different machines, or different magma processes on the same machine, or even in the same magma process.

We begin with process A:

```
> R<X,Y> := PolynomialRing(Integers( ),2);
> n := 704787824974797479872349583 41;
> F := 814*X^4 + 3172*X^3*Y - 49218*X^2*Y^2 - 142775*X*Y^3
>     - 65862*Y^4;
> m1 := 3050411;
> m2 := 1;
> A := NFSProcess(n,F,m1,m2);
> A`Firstb := 0;
> A`Lastb := 99;
> A`OutputFilename := "/tmp/nfs-distrib-A";
> FindRelations(A);
3852
```

Now, process B, with n, F, $m1$, and $m2$ as above:

```
> B := NFSProcess(n,F,m1,m2);
> B`Firstb := 99;
> B`Lastb := 199;
> B`OutputFilename := "/tmp/nfs-distrib-B";
> FindRelations(B);
2455
```

Then later, on a single machine,

```
> input_files := ["/tmp/nfs-distrib-A","/tmp/nfs-distrib-B"];
> P := NFSProcess(n,F,m1,m2);
> P`OutputFilename := "/tmp/nfs-distrib-all";
> MergeFiles(input_files, P`OutputFilename);
4162 25925
> CycleCount(P);
4368
> CreateCycleFile(P);
> CreateCharacterFile(P);
> FindDependencies(P);
```

Now, the final factorization stage may be distributed over more than one processor also. We attempt to factor a relation on A:

```
> A`OutputFilename := "/tmp/nfs-distrib-all";
> Factor(A,9);  // factor dependency 9
0
```

No factor was found on machine A, but meantime on B:

```
> B`OutputFilename := "/tmp/nfs-distrib-all";
> Factor(P,1);  // factor dependency 1
94899629
> n mod $1, n div $1;
0 742666575624650207929
```

We have a successful factorisation.

19.14.6 Magma and CWI NFS Interoperability

At the time of writing, the record NFS factorizations were lead by the CWI group and by people using CWI's or Arjen Lenstra's code. The CWI tools use a different data file format to Magma's native format, but Magma supplies some tools to allow users to assist in CWI factorization attempts.

The user may generate relations in CWI relation format, rather than Magma native format, by using `FindRelationsInCWIFormat`. The user should note that relations in CWI format cannot at present be used in the Auxiliary data, Linear algebra, or Factorization stages of the Magma NFS.

Alternatively, assuming some Magma NFS relations have already been computed for a process, then the user may use the procedure `ConvertToCWIFormat` to convert the relation data files from Magma native format to CWI format. The resulting data file is named `OutputFilename`_CWI_format, and will contain both the full and partial relations of the process.

`FindRelationsInCWIFormat(P): NFSProc -> RngIntElt`

Given an NFS process P for factoring an integer n, generates relations to factor n with the Number Field Sieve algorithm, in a file format suitable for use with CWI's NFS tools. Returns the number of relations found.

`ConvertToCWIFormat(P, pb): NFSProc, RngIntElt -> .;`

Converts the relation files of the NFS process P to CWI format, storing primes only greater than or equal to the prime printing bound *pb*. The resulting data file name will be named `P`OutputFilename`_CWI_format.

19.14.7 Tools for Finding a Suitable Polynomial

Magma does not provide a function to select an optimal polynomial for the factorization of a given number. However, Magma does provide some functions that are useful for the implementation of the polynomial selection algorithms developed by Peter Montgomery and Brian Murphy in [Mur99].

The functions `BaseMPolynomial`, `MurphyAlphaApproximation`, `OptimalSkewness`, `BestTranslation`, `PolynomialSieve`, and `DickmanRho`, will be useful for those wanting to implement polynomial selection routines within the Magma interpreter language.

`BaseMPolynomial(n, m, d): RngIntElt, RngIntElt, RngIntElt -> RngMPolElt`

Given integers n, m, and d, returns a homogeneous bivariate polynomial $F = \sum_{i=0}^{d} c_i X^i Y^{d-i}$ such that the coefficients c_i give a base m representation of n: that is, $\sum_{i=0}^{d} c_i m^i = n$. The coefficients also satisfy $|c_i| \leq m/2$.

This polynomial F may be used to factorize n using the number field sieve (with $m_1 := m$ and $m_2 := 1$).

This function requires that $d \geq 2$ and $n \geq m^d$.

`MurphyAlphaApproximation(F, b): RngMPolElt, RngIntElt -> FldReElt`

Given a univariate or homogeneous bivariate polynomial F, return an approximation of the α value of F, using primes less than the positive integer bound b.

The α value of a polynomial is defined in [Mur99].

Since random sampling is used for primes dividing the discriminant, successive calls to this function will give slightly different results.

`OptimalSkewness(F): RngMPolElt -> FldReElt, FldReElt`

Given a univariate or homogeneous bivariate polynomial F, return its optimal skewness and corresponding average log size.

The optimal skewness and average log size values are defined in [Mur99].

Example: Finding a Good NFS Polynomial `(ex-112b3b)`

This example illustrates an effective (though not optimal) method for finding a "good" polynomial for use in NFS factorizations.

Here we search for a degree $d = 4$ polynomial to use in factoring a 52-digit integer n.

We define the rating of a polynomial to be the sum of the α value and corresponding "average log size" (see [Mur99]).

We then proceed by iterating over base m polynomials with successive leading coefficients (with the values of m near $(m^{1/d}m^{1/d+1})^{1/2}$, and chosen to minimize the second-to-leading coefficient), and choosing as a result the polynomial with the smallest rating.

```
> n := RandomPrime(90)*RandomPrime(90);
> n;
359635470725625320407673937416777014871521894980388 9
> d := 4;
> approx_m := Iroot( Iroot( n, d+1 ) * Iroot( n, d ) , 2 );
> leading_coeff := n div approx_m^d;
> leading_coeff;
143082
> m := Iroot( n div leading_coeff, d );
> P<X,Y> := PolynomialRing( Integers(), 2 );
> F<X,Y> := BaseMPolynomial(n,m,d);
> F;
143082*X^4 + 463535*X^3*Y - 173869838910*X^2*Y^2 + 167201617413*X*Y^3 +
    159859288415*Y^4
> skew, als := OptimalSkewness( F );
> alpha := MurphyAlphaApproximation( F, 2000 );
> rating := als + alpha;
> rating;
23.143714548914575193314917
>
> best_rating := rating;
> best_m := m;
> for i in [1..100] do
>   leading_coeff := leading_coeff + 1;
>   m := Iroot( n div leading_coeff, d );
>   F<X,Y> := BaseMPolynomial(n,m,d);
>   skew, als := OptimalSkewness( F );
>   alpha := MurphyAlphaApproximation( F, 2000 );
>   rating := als + alpha;
>   if rating lt best_rating then
>     best_rating := rating;
>     best_m := m;
```

(continues on next page)

(continued from previous page)

```
>   end if;
> end for;
> best_rating;
20.899568473033257031950385
> best_m;
398116527578
> F<X,Y> := BaseMPolynomial(n,best_m,d);
> F;
143160*X^4 + 199085*X^3*Y - 9094377652*X^2*Y^2 - 93898749030*X*Y^3 -
    169859083883*Y^4
> OptimalSkewness( F );
165.514255523681640625 20.96993446792061218064 6408
> MurphyAlphaApproximation( F, 2000 );
-0.05427161576301414495001508 42
> time NFS( n, F, best_m, 1 );
...
```

BestTranslation(F, m, a): RngMPolElt, RngIntElt, FldReElt, FldReElt -> RngMPolElt, RngIntElt, FldReElt, FldReElt

Given a univariate or homogeneous bivariate polynomial F, an integer m, and real value a (which should be the average log size of F for some optimal skewness), returns a polynomial G and an integer m' such that $G(m') = F(m)$, together with the average log size and optimal skewness of G. The translation G is selected such that the average log size is a local minimum.

PolynomialSieve(F, m, J0, J1, MaxAlpha): RngMPolElt, RngIntElt, RngIntElt, RngIntElt, FldReElt -> List

```
PrimeBound: RngIntElt                    Default: 1000
```

Given a homogeneous bivariate integer polynomial F of degree d, together with integers m, J_0 and J_1 and a real value `MaxAlpha`, returns a list of tuples, each of which contains a polynomial $G = F + j_1 x^2 y^{d-2} - (|j_0| + j_1 m)xy^{d-1} + (j_0 m)y^d$, where $|j_0| \leq J_0$ and $|j_1| \leq J_1$ such that the α value (see [Mur99]) of G is "better" (that is, lower) than `MaxAlpha`.

Each tuple contains the data <average log size + α, skewness, α, G, m, j_0, j_1 >.

If the optional parameter `PrimeBound` is set, it is used as an upper bound for primes used to calculate α.

19.15 Bibliography

- [AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
- [AM93] A. O. L. Atkin and F. Morain. Elliptic curves and primality proving. *Math. Comp.*, 61:29 – 68, 1993.
- [Bre80] R. P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20:176–184, 1980.
- [BtR92] R. P. Brent and H. J. J te Riele. Factorizations of $a^n \pm 1$, $13 \leq a < 100$. Technical Report, Centrum voor Wiskunde en Informatica, Amsterdam, 1992. URL:ftp://nimbus.anu.edu.au/pub/Brent.
- [FKMW04] J. Franke, T. Kleinjung, F. Morain, and T. Wirth. Proving the primality of very large numbers with fastECPP. In *Algorithmic number theory*, volume 3076 of Lecture Notes in Comput. Sci., pages 194–207. Springer, Berlin, 2004.
- [Knu97] Donald E. Knuth. *The Art of Computer Programming*. Volume 2. Addison Wesley, Reading, Massachusetts, 3rd edition, 1997.

- [LD95] Arjen K. Lenstra and Bruce Dodson. NFS with four large primes: An explosive experiment. In Don Coppersmith, editor, *Advances in cryptology—CRYPTO 1995*, volume 963 of LNCS, 372–385. Berlin, 1995. Springer.
- [Mar95] G. Marsaglia. DIEHARD: a battery of tests of randomness. http://stat.fsu.edu/pub/diehard/, 1995.
- [Mar00] G. Marsaglia. The Monster, a random number generator with period over 10^{2857} times as long as the previously touted longest-period one. Preprint, 2000.
- [Mar92] George Marsaglia. The mathematics of random number generators. In *The unreasonable effectiveness of number theory (Orono, ME, 1991)*, volume 46 of Proc. Sympos. Appl. Math., pages 73–90. Amer. Math. Soc., Providence, RI, 1992. URL: http://dx.doi.org/10.1090/psapm/046/1195842.
- [MZ91] George Marsaglia and Arif Zaman. A new class of random number generators. *Ann. Appl. Probab.*, 1(3):462–480, 1991.
- [Mon92] Peter Lawrence Montgomery. *An FFT Extension of the Elliptic Curve Method of Factorization*. PhD thesis, University of California, Los Angeles, 1992.
- [Mur99] Brian Murphy. *Polynomial selection for the number field sieve integer factorisation algorithm*. PhD thesis, Oxford University, 1999. http://web.comlab.ox.ac.uk/oucl/work/richard.brent/ftp/Murphy-thesis.ps.gz.
- [Sch71] Arnold Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.
- [SW17] Jonathan Sorenson and Jonathan Webster. Strong Pseudoprimes to Twelve Prime Bases. *Mathematics of Computation*, 86(304):985–1003, 2017. doi:10.1090/mcom/3134.
- [vzGG99] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, 1999.
- [Web95] Kenneth Weber. The Accelerated Integer GCD Algorithm. *ACM Transactions on Mathematical Software*, 21(1):111–122, 1995.

Chapter 20

Integer Residue Class Rings

20.1 Introduction

This chapter presents the machinery provided in Magma for computing in quotient rings of the ring of integers $\mathbb{Z}$, that is, integer residue class rings. The first half of the chapter describes operations with ideals of $\mathbb{Z}$ and their quotient rings while the second half provides an introduction to computing with Dirichlet characters.

20.2 Ideals of $\mathbb{Z}$

The theory of ideals of $\mathbb{Z}$ is very elementary but for completeness the general machinery for ring ideals applies. Such ideals will have type `RngInt`, that is, the same type as the ring of integers itself (`ideal<Integers() | 1>`).

In the case of $\mathbb{Z}$ any subring is an ideal so that the `sub`-constructor creates the same object as does the `ideal`-constructor. One quirk of this fact that ideals are of type `RngInt` is that taking R/I will result in the *ideal division* rather than the ring-by-ideal quotient (this is for compatibility with orders of number fields, as explained in the next section). Here is an example that shows the difference.

```
> Z := Integers();
> I := ideal<Z|1>; // ideal of Z
> Z/I; // interpreted as ideal division
Integer Ring
> quo<Z|I>; // quotient of ring by ideal
Residue class ring of integers modulo 1
```

`ideal< R | a >: RngInt, RngIntElt -> RngIntRes`

Given the ring of integers $\mathbb{Z}$ and an integer a, return the ideal of $\mathbb{Z}$ generated by a.

Example: Ideals of the Integers `(ex-1aa6f6)`

We construct some ideals of $\mathbb{Z}$.

```
> Z := IntegerRing();
> I13 := ideal< Z | 13 >;
> I13;
Ideal of Integer Ring generated by 13
> 1 in I13;
false
> 0 in I13;
true
> -13 in I13;
```

(continues on next page)

(continued from previous page)

```
true
> I0 := ideal< Z | 0 >;
> 0 in I0;
true
> 1 in I0;
false
```

We check that $\mathbb{Z}$ is regarded as an ideal.

```
> I1 := ideal< Z | 1 >;
> I1 eq Z;
true
```

20.3 $\mathbb{Z}$ as a Number Field Order

A collection of functions are provided that make $\mathbb{Z}$ behave like an order of a number field. Note however, that $\mathbb{Z}$ is not of type `RngOrd`. If complete compatibility is necessary, the user should create the maximal order of a degree 1 extension of $\mathbb{Q}$.

`Decomposition(R, p): RngInt, RngIntElt -> SeqEnum`
Returns the ideal decomposition of the prime p, i.e. a list `[ < ideal<Z|p>, 1> ]` as in the number field case.

`Generator(I): RngInt -> RngIntElt`
A generator for the given ideal.

`RamificationIndex(I, p): RngInt, RngIntElt -> RngIntElt`
`RamificationIndex(I): RngInt -> RngIntElt`
The ramification index of I over $\mathbb{Z}$ which is always 1.

`Degree(I): RngInt -> RngIntElt`
The inertia degree of the ideal I, which is always 1.

`TwoElementNormal(I): RngInt -> RngIntElt, RngIntElt`
Two integers that generate the ideal I. In this case the generator is returned twice.

`ChineseRemainderTheorem(I, J, a, b): RngInt, RngInt, RngIntElt, RngIntElt -> RngIntElt`
The Chinese remainder theorem for ideals. Given ideals I and J of $\mathbb{Z}$ together with integers a and b, an integer x such that $x - a \in I$ and $x - b \in J$ is returned.

`Valuation(x, I): RngIntElt, RngInt -> RngIntElt`
The valuation of the integer x at the prime ideal I.

`ClassRepresentative(I): RngInt -> RngInt`
The representative of the ideal I of $\mathbb{Z}$ in the basis of the class group.

20.4 Residue Class Rings

The ring $\mathbb{Z}/m\mathbb{Z}$ consists of representatives for the residue classes of integers modulo $m > 1$. This Section describes the operations in Magma for such rings and their elements.

At any stage during a session, Magma will have at most one copy of $\mathbb{Z}/m\mathbb{Z}$ present, for any $m > 1$. In other words, different names for the same residue class ring will in fact be different references to the same structure. This saves memory and avoids confusion about different but isomorphic structures.

If m is a prime number, the ring $\mathbb{Z}/m\mathbb{Z}$ forms a field; however, Magma has special functions for dealing with finite fields. The operations described here should *not* be used for finite field calculations: the implementation of finite field arithmetic in Magma as described in Chapter *Finite Fields* takes full advantage of the special structure of finite fields and leads to superior performance.

20.4.1 Creation

In addition to the general quotient constructor, a number of abbreviations are provided for computing residue class rings.

`quo<Z | I>: RngInt, RngInt -> RngIntRes`

Given the ring of integers $\mathbb{Z}$, and an ideal I, create the residue class ring modulo the ideal.

Note that Z/I does **not** give this residue ring, but rather, in compatibility with $\mathbb{Z}$ as a number field order, returns the ideal quotient, namely $\mathbb{Z}$ itself (see Section *{\mathbb{Z}} as a Number Field Order* above, and the example in Sections *Ideals of {\mathbb{Z}}*).

`quo<Z | m>: RngInt, RngIntElt -> RngIntRes`

Given the ring of integers $\mathbb{Z}$, and an integer $m \neq 0$, create the residue class ring $\mathbb{Z}/m\mathbb{Z}$.

`ResidueClassRing(m): RngIntElt -> RngIntRes, Map`

`IntegerRing(m): RngIntElt -> RngIntRes`

`Integers(m): RngIntElt -> RngIntRes`

`RingOfIntegers(m): RngIntElt -> RngIntRes`

Given an integer greater than zero, create the residue class ring $\mathbb{Z}/m\mathbb{Z}$ and also returns the map from $\mathbb{Z}$ into $\mathbb{Z}/m\mathbb{Z}$.

`ResidueClassField(p): RngIntElt -> FldFin, Map`

Given a prime integer p construct the residue class field $\mathbb{F}_p$ and the map from $\mathbb{Z}$ into $\mathbb{F}_p$.

`ResidueClassRing(Q): RngIntEltFact -> RngIntRes`

`IntegerRing(Q): RngIntEltFact -> RngIntRes`

`Integers(Q): RngIntEltFact -> RngIntRes`

Create the residue class ring $\mathbb{Z}/m\mathbb{Z}$, where m is the integer corresponding to the factorization sequence Q. This is more efficient than creating the ring by m alone, since the factorization Q will be stored so it can be reused later.

Example: Cube Root in a Prime Residue Ring (ex-d9f52c)

We construct a residue ring having modulus the largest prime not exceeding 2^{16}.

```
> p := PreviousPrime(2^16);
> p;
65521
> R := ResidueClassRing(p);
Residue class ring of integers modulo 65521
```

Now we try to find an element x in R such that $x^3 = 23$.

```
> exists(t){x : x in R | x^3 eq 23};
true
> t;
12697
```

20.4.2 Coercion

As can be seen from the tables in Chapter *Introduction to Rings*, automatic coercion takes place between $\mathbb{Z}/m\mathbb{Z}$ and $\mathbb{Z}$ so that a binary operation like + applied to an element of $\mathbb{Z}/m\mathbb{Z}$ and an integer will result in a residue class from $\mathbb{Z}/m\mathbb{Z}$. Using `!`, elements from a prime field $\mathbf{F}_p$ can be coerced into $\mathbb{Z}/p\mathbb{Z}$, and elements from $\mathbb{Z}/p\mathbb{Z}$ can be coerced into $\mathbf{F}_{p^r}$. Also, transitions between $\mathbb{Z}/m\mathbb{Z}$ and $\mathbb{Z}/n\mathbb{Z}$ can be made using `!` provided that m divides n or n divides m. In cases where there is a choice – such as when an element r from $\mathbb{Z}/m\mathbb{Z}$ is coerced into $\mathbb{Z}/n\mathbb{Z}$ with m dividing n – the result will be the residue class containing the representative for r.

Example: Coercion between Residue Class Rings (ex-155ae3)

```
> r := ResidueClassRing(3) ! 5;
> r;
2
> ResidueClassRing(6) ! r;
2
```

So the representative 2 of 5 mod 3 is mapped to the residue class 2 mod 6, and not to 5 mod 6.

20.4.3 Elementary Invariants

`Characteristic(R): RngIntRes -> RngIntResElt`

`# R: RngIntRes -> RngIntResElt`

`Modulus(R): RngIntRes -> RngInt`

Given a residue class ring $R = \mathbb{Z}/m\mathbb{Z}$, this function returns the common modulus m for the elements of R.

`FactoredModulus(R): RngIntRes -> RngIntEltFact`

Given a residue class ring $R = \mathbb{Z}/m\mathbb{Z}$, this function returns the factorization of the common modulus m for the elements of R.

20.4.4 Structure Operations

`AdditiveGroup(R): RngIntRes -> GrpAb, Map`

Given $R = \mathbb{Z}/m\mathbb{Z}$, create the abelian group of integers modulo m under addition. This returns the finite additive abelian group A (of order m) together with a map from A to the ring $\mathbb{Z}/m\mathbb{Z}$, sending $A.1$ to 1.

`MultiplicativeGroup(R): RngIntRes -> GrpAb, Map`

`UnitGroup(R): RngIntRes -> GrpAb, Map`

Given $R = \mathbb{Z}/m\mathbb{Z}$, create the multiplicative group of R as an abelian group. This returns an (additive) abelian group A of order $\phi(m)$, together with a map from A to R.

`sub< R | n >: RngIntRes, RngIntResElt -> RngIntRes`

Given R, the ring of integers modulo m or an ideal of it, and an element n of R create the ideal of R generated by n.

`Set(R): RngIntRes -> SetEnum`

Create the enumerated set consisting of the elements of the residue class ring R.

`Category(R): RngIntRes -> Cat`

`Parent(R): RngIntRes -> PowerStructure`

`PrimeRing(R): RngIntRes -> RngIntRes`

`Center(R): RngIntRes -> RngIntRes`

20.4.5 Ring Predicates and Booleans

`IsCommutative(R): RngIntRes -> BoolElt`

`IsUnitary(R): RngIntRes -> BoolElt`

`IsFinite(R): RngIntRes -> BoolElt`

`IsOrdered(R): RngIntRes -> BoolElt`

```
IsField(R): RngIntRes -> BoolElt
```

```
IsEuclideanDomain(R): RngIntRes -> BoolElt
```

```
IsPID(R): RngIntRes -> BoolElt
```

```
IsUFD(R): RngIntRes -> BoolElt
```

```
IsDivisionRing(R): RngIntRes -> BoolElt
```

```
IsEuclideanRing(R): RngIntRes -> BoolElt
```

```
IsPrincipalIdealRing(R): RngIntRes -> BoolElt
```

```
IsDomain(R): RngIntRes -> BoolElt
```

```
R eq R: RngIntRes, Rng -> BoolElt
```

```
R ne R: RngIntRes, Rng -> BoolElt
```

20.4.6 Homomorphisms

Ring homomorphisms with domain $\mathbb{Z}/m\mathbb{Z}$ are completely determined by the image of 1. As usual (see Chapter *Ring of Integers*), we require our homomorphisms to map 1 to 1. Therefore, the general homomorphism constructor with domain $\mathbb{Z}/m\mathbb{Z}$ needs no arguments.

```
hom< R -> S | >: RngIntRes, Rng -> Map
```

Given a residue class ring R, and a ring S, create a homomorphism from R to S, determined by $f(1_R) = 1_S$. Note that it is the responsibility of the user that the map defines a homomorphism!

20.5 Elements of Residue Class Rings

20.5.1 Creation

```
elt< R | k >: RngIntRes, RngIntElt -> RngIntResElt
```

Create the residue class containing the integer k in residue class ring R.

```
R ! k: RngIntRes, RngIntElt -> RngIntResElt
```

Create the residue class containing k in the residue class ring R. Here k is allowed to be either an integer, or an element of the finite field $\mathbb{F}_p$ in the case $R = \mathbb{Z}/p\mathbb{Z}$, or an element of $S = \mathbb{Z}/n\mathbb{Z}$ for a multiple or divisor n of m (with $R = \mathbb{Z}/m\mathbb{Z}$).

```
One(R): RngIntRes -> RngIntResElt
Identity(R): RngIntRes -> RngIntResElt
Zero(R): RngIntRes -> RngIntResElt
Representative(R): RngIntRes -> RngIntResElt
```

These generic functions (cf. Chapter *Introduction to Rings*) create 1, 1, 0, and 0 respectively, in any $\mathbb{Z}/m\mathbb{Z}$.

```
Random(R): RngIntRes -> RngIntResElt
```

Create a "random" residue class in R.

20.5.2 Arithmetic Operators

```
+ n: RngIntResElt -> RngIntResElt
```

```
- n: RngIntResElt -> RngIntResElt
```

```
m + n: RngIntResElt, RngIntResElt -> RngIntResElt
```

```
m - n: RngIntResElt, RngIntResElt -> RngIntResElt
m * n: RngIntResElt, RngIntResElt -> RngIntResElt
n ^ k: RngIntResElt, RngIntResElt -> RngIntResElt
m / n: RngIntResElt, RngIntResElt -> RngIntResElt
m div n: RngIntResElt, RngIntResElt -> RngIntResElt
m +:= n: RngIntResElt, RngIntResElt -> RngIntResElt
m -:= n: RngIntResElt, RngIntResElt -> RngIntResElt
m *:= n: RngIntResElt, RngIntResElt -> RngIntResElt
m /:= n: RngIntResElt, RngIntResElt -> RngIntResElt
m ^:= k: RngIntResElt, RngIntResElt -> RngIntResElt
```

20.5.3 Equality and Membership

```
m eq n: RngIntResElt, RngIntResElt -> BoolElt
m ne n: RngIntResElt, RngIntResElt -> BoolElt
n in R: RngIntResElt, Rng -> BoolElt
n notin R: RngIntResElt, Rng -> BoolElt
```

20.5.4 Parent and Category

```
Parent(n): RngIntResElt -> RngIntRes
Category(n): RngIntResElt -> Cat
```

20.5.5 Predicates on Ring Elements

```
IsZero(n): RngIntResElt -> BoolElt
IsOne(n): RngIntResElt -> BoolElt
IsMinusOne(n): RngIntResElt -> BoolElt
IsNilpotent(n): RngIntResElt -> BoolElt
IsIdempotent(n): RngIntResElt -> BoolElt
IsUnit(n): RngIntResElt -> BoolElt
IsZeroDivisor(n): RngIntResElt -> BoolElt
IsRegular(n): RngIntRes -> BoolElt
IsIrreducible(n): RngIntResElt -> BoolElt
IsPrime(n): RngIntResElt -> BoolElt
```

20.5.6 Solving Equations over $\mathbb{Z}/m\mathbb{Z}$

`Solution(a, b): RngIntResElt, RngIntResElt -> RngIntResElt`

Given elements a and b of $\mathbb{Z}/m\mathbb{Z}$, return a solution x to the linear congruence $a \cdot x = b \in \mathbb{Z}/m\mathbb{Z}$. An error is signalled if no solution exists.

`IsSquare(n): RngIntResElt -> BoolElt, RngIntResElt`

Factorization: [<RngIntElt, RngIntElt>]	Default: [ ]

Given an element $n \in \mathbb{Z}/m\mathbb{Z}$ this function returns `true` if there exists $a \in \mathbb{Z}/m\mathbb{Z}$ such that $a^2 = n \in \mathbb{Z}/m\mathbb{Z}$, `false` otherwise. If n is a square, a square root a is also returned. If m is large and its prime factorization is known, the computation may be speeded up by assigning the factorization sequence for m to the optional argument `Factorization`.

`Sqrt(a): RngIntResElt -> RngIntResElt`

`SquareRoot(a): RngIntResElt -> RngIntResElt`

Factorization: [<RngIntElt, RngIntElt>]	Default: [ ]

Given an element a of the ring $\mathbb{Z}/m\mathbb{Z}$, this function returns an element b of $\mathbb{Z}/m\mathbb{Z}$ such that $b^2 = a \in \mathbb{Z}/m\mathbb{Z}$, if such an element exists, and an error otherwise. If m is large and its prime factorization is known, the computation may be speeded up by assigning the factorization sequence for m to the optional argument `Factorization`.

`AllSquareRoots(a): RngIntResElt -> [ RngIntResElt ]`

`AllSqrts(a): RngIntResElt -> [ RngIntResElt ]`

Factorization: [<RngIntElt, RngIntElt>]	Default: [ ]

Return a sequence containing all square roots of the element a in a residue class ring $\mathbb{Z}/m\mathbb{Z}$. If the modulus m is large and its prime factorization is known, the computation may be speeded up by assigning the factorization sequence for m to the optional argument `Factorization`.

Example: Square Roots Modulo 2340 `(ex-c52d75)`

We construct the residue class ring having modulus 2340 and find all the square roots of 1404.

```
> R := ResidueClassRing(2340);
Residue class ring of integers modulo 2340
> x := R!1404;
> sqrts := AllSquareRoots(x);
> sqrts;
[ 78, 312, 468, 702, 858, 1092, 1248, 1482, 1638,
  1872, 2028, 2262 ]
> [ y^2 : y in sqrts ];
[ 1404, 1404, 1404, 1404, 1404, 1404, 1404, 1404,
  1404, 1404, 1404, 1404 ]
```

So 1404 has 12 square roots!

20.6 Ideal Operations

`ideal< R | a1, ..., ar >: RngIntRes, RngIntResElt, ..., RngIntResElt -> RngIntRes`

The ideal of the residue ring R generated by the greatest common divisor of the elements a_i and the modulus of R.

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

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

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

Greatest common divisor of the elements a and b of R, that is, a generator for the R-ideal $(a)+(b)$.

`GreatestCommonDivisor(Q): [RngIntResElt] -> RngIntResElt`

`Gcd(Q): [RngIntResElt] -> RngIntResElt`

`GCD(Q): [RngIntResElt] -> RngIntResElt`

Greatest common divisor of the sequence of elements Q, that is, a generator for the R-ideal generated by the elements in Q.

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

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

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

Least common multiple of the elements a and b of R, that is, a generator for the R-ideal $(a) \cap (b)$.

`LeastCommonMultiple(Q): [RngIntResElt] -> RngIntResElt`

`Lcm(Q): [RngIntResElt] -> RngIntResElt`

`LCM(Q): [RngIntResElt] -> RngIntResElt`

Least common multiple of the sequence of elements Q, that is, a generator for the R-ideal formed by the intersection of the principal ideals generated by elements of Q.

`I + J: RngIntRes, RngIntRes -> RngIntRes`

`I * J: RngIntRes, RngIntRes -> RngIntRes`

`I meet J: RngIntRes, RngIntRes -> RngIntRes`

`a in I: RngIntResElt, RngIntRes -> BoolElt`

`a notin I: RngIntResElt, RngIntRes -> BoolElt`

`I eq J: RngIntRes, RngIntRes -> BoolElt`

`I ne J: RngIntRes, RngIntRes -> BoolElt`

`I subset J: RngIntRes, RngIntRes -> BoolElt`

`I notsubset J: RngIntRes, RngIntRes -> BoolElt`

20.7 The Unit Group

`UnitGroup(R): RngIntRes -> GrpAb, Map`

Given $R = \mathbb{Z}/m\mathbb{Z}$, construct the unit group of R as an abelian group. This returns an (additive) abelian group A of order $\phi(m)$, together with a map from A to R.

`IsPrimitive(n): RngIntResElt -> BoolElt`

Returns `true` if the element $n \in \mathbb{Z}/m\mathbb{Z}$ is primitive, that is, if it generates the multiplicative group of $\mathbb{Z}/m\mathbb{Z}$, `false` otherwise.

`PrimitiveElement(R): RngIntRes -> RngIntResElt`

`PrimitiveRoot(R): RngIntRes -> RngIntResElt`

Given $R = \mathbb{Z}/m\mathbb{Z}$, this function returns a generator for the group of units of R if this group is cyclic, and returns 0 otherwise. Thus a valid generator is only returned if $m = 2, 4, p^t$ or $2p^t$, with p an odd prime and $t \geq 1$.

`Order(a): RngIntResElt -> RngIntElt`

Given an element a belonging to $\mathbb{Z}/m\mathbb{Z}$, return the multiplicative order $k \geq 1$ of a if a is in the unit group $(\mathbb{Z}/m\mathbb{Z})^*$, and zero if a is not a unit.

`Normalize(x): RngIntRes -> RngIntResElt, RngIntResElt`

`Normalise(x): RngIntRes -> RngIntResElt, RngIntResElt`

Given an element $x \in R = \mathbb{Z}/m\mathbb{Z}$, this function returns the unique canonical associate $y \in R$ of x and a unit $u \in R$ such that $u \cdot x = y$. The canonical associate of x is the GCD of x and m, considered as natural integers (unless x is 0, in which case it is 0).

Example: Unit Group of a Residue Class Ring (ex-8ae5da)

We determine the unit group of the ring with modulus 735 and then verify its order by comparing it with $\phi(m)$.

```
> m := 735;
> R := ResidueClassRing(m);
Residue class ring of integers modulo 735
> U, psi := UnitGroup(R);
> U;
Abelian Group isomorphic to Z/2 + Z/2 + Z/84
Defined on 3 generators
Relations:
    2*U.1 = 0
    4*U.2 = 0
    42*U.3 = 0
> #U;
336
> EulerPhi(735);
336
```

So the order of U is equal to $\phi(m)$ as it should be. Finally, we look for three elements of R that generate the unit group.

```
> gens := [ psi(U.i) : i in [1..3] ]; gens;
> [ Order(x) : x in gens ];
[ 2, 4, 42 ]
```

Example: Cyclic Unit Group (ex-bde8d2)

We construct a residue class ring $R = \mathbb{Z}/m\mathbb{Z}$ having cyclic unit group. By a theorem of Gauss, the ring R has cyclic unit group precisely when $n = 4, n = p^e$, or $n = 2p^e$, and p is an odd prime.

```
> R := IntegerRing(50);
> U, psi := UnitGroup(R);
Abelian Group isomorphic to Z/20
Defined on 1 generator
Relations:
    20*U.1 = 0
> w := PrimitiveElement(R);
> w;
3
> Order(w);
20
```

We verify that the powers of w are precisely the elements of the unit group U.

```
> powers := { w^i : i in [0..19] };
> powers;
{ 29, 1, 31, 3, 33, 7, 37, 9, 39, 11, 41, 13, 43, 17, 47, 19, 49, 21, 23, 27 }
> powers eq { psi(u) : u in U };
true
```

20.8 Dirichlet Characters

Let R be a ring. Then a *Dirichlet character over R of modulus N* is a homomorphism

$$\varepsilon : (\mathbb{Z}/N\mathbb{Z})^* \to R^*,$$

where R^* is the group of invertible elements of R. We extend ε to a set theoretic map on the whole of $\mathbb{Z}$ by defining $\varepsilon(x) = 0$ if $\gcd(x, N) \neq 1$. The *conductor* of ε is the smallest positive integer M such that the homomorphism $(\mathbb{Z}/N\mathbb{Z})^* \to R^*$ factors through $(\mathbb{Z}/M\mathbb{Z})^*$ via the natural map $(\mathbb{Z}/N\mathbb{Z})^* \to (\mathbb{Z}/M\mathbb{Z})^*$.

20.8.1 Creation

`DirichletGroup(N): RngIntElt -> GrpDrch`

The group of Dirichlet characters modulo N with values in `RationalField()`. (Note that this is a group of exponent at most 2.)

`FullDirichletGroup(N): RngIntElt -> GrpDrch`

The group of Dirichlet characters modulo N taking values in the mth cyclotomic field, where m is the exponent of the unit group modulo N.

`DirichletGroup(N, R): RngIntElt, Rng -> GrpDrch`

The group of Dirichlet characters modulo N with values in the ring R. Here R can be the integers, rationals, a number field, or a finite field. (Note that this group may be smaller than the full Dirichlet group.)

`DirichletGroup(N, R, z, r): RngIntElt, Rng, RngElt, RngIntElt -> GrpDrch`

The group of Dirichlet characters mod N with values in the cyclic subgroup generated by the root of unity z in the ring R. Here z must be an element of R of exact order r (where r may be smaller than the exponent of the full Dirichlet group).

`BaseExtend(G, R): GrpDrch, Rng -> GrpDrch`

`BaseExtend(G, R, z): GrpDrch, Rng, RngElt -> GrpDrch`

The group of Dirichlet characters corresponding to G with values in the ring R. In the second form, the distinguished root of unity of the base ring of G is identified with the given element z.

`AssignNames(~G, S): GrpDrch, [MonStgElt]`

Assign names to the generators of the Dirichlet group G.

20.8.2 Element Creation

`Elements(G): GrpDrch -> [GrpDrchElt]`

A sequence containing all Dirichlet characters in the Dirichlet group G.

`Random(G): GrpDrch -> GrpDrchElt`

A random element of the Dirichlet group G.

`G . i: GrpDrch, RngIntElt -> GrpDrchElt`

The ith generator of the group G.

`G ! x: GrpDrch, Any -> GrpDrchElt`

This coerces the given element x into the Dirichlet group G. Here x may be a Dirichlet character belonging to a different group, or a sequence of integers specifying an element of the `AbelianGroup` of G.

`KroneckerCharacter(D): RngIntElt -> GrpDrchElt`

`KroneckerCharacter(D, R): RngIntElt, Rng -> GrpDrchElt`

The Kronecker character $n \mapsto (d/n)$, where d is the fundamental discriminant associated to the integer D.

When a ring R is given, this is returned as a character with values in R.

20.8.3 Attributes of Dirichlet Groups

`BaseRing(G): GrpDrch -> Rng`

The ring in which characters in G take values.

`Modulus(G): GrpDrch -> RngIntElt`

The integer N such that G is a group of Dirichlet characters on $\mathbb{Z}/N$.

`Order(G): GrpDrch -> RngIntElt`

The order of the Dirichlet group G.

`Exponent(G): GrpDrch -> RngIntElt`

The exponent of the Dirichlet group G.

`NumberOfGenerators(G): GrpDrch -> RngIntElt`

The number of generators of the Dirichlet group G.

`Generators(G): GrpDrch -> [GrpDrchElt]`

A sequence containing generators for the Dirichlet group G.

`UnitGenerators(G): GrpDrch -> [RngIntElt]`

This returns an ordered sequence of integers that reduce to "canonical" generators of the unit group of Z/N, where N is the modulus of G.

`GaloisConjugacyRepresentatives(G): GrpDrch -> [GrpDrchElt]`

`GaloisConjugacyRepresentatives(seq): [GrpDrchElt] -> [GrpDrchElt]`

This returns a sequence containing one representative from each Galois conjugacy class (over $\mathbb{Q}$) of characters corresponding to a character in the given group or the given sequence.

`AbelianGroup(G): GrpDrch -> GrpAb, Map`

This returns a finite abelian group isomorphic to the given group G of Dirichlet characters (as an abstract group), and secondly returns a map from the abstract group to G.

It is necessary to use this function in order to make group theoretic constructions involving G.

20.8.4 Attributes of Elements

`BaseRing(chi): GrpDrchElt -> Rng`

The ring in which the Dirichlet character χ takes values.

`Modulus(chi): GrpDrchElt -> RngIntElt`

The modulus of the group of Dirichlet characters that contains χ.

`Conductor(chi): GrpDrchElt -> RngIntElt`

The minimal conductor of the Dirichlet character χ. (That is, the smallest integer M such that `chi` is well-defined on the unit group of Z/M.)

`ElementToSequence(chi): GrpDrchElt -> SeqEnum`

A sequence of integers specifying the Dirichlet character χ (in terms of generators of the group containing χ).

`x eq y: GrpDrchElt, GrpDrchElt -> BoolElt`

Return `true` iff the given characters have the same modulus and values.

`Order(chi): GrpDrchElt -> RngIntElt`

The order of the given element χ in a group of Dirichlet characters.

`IsTrivial(chi): GrpDrchElt -> BoolElt`

Returns `true` if and only if the Dirichlet character χ has order 1.

`IsPrimitive(chi): GrpDrchElt -> BoolElt`

Returns `true` iff the Dirichlet character χ is primitive (equivalently, if its conductor equals its modulus).

`AssociatedPrimitiveCharacter(chi): GrpDrchElt -> GrpDrchElt`

The primitive character modulo the conductor of χ which takes the same values (on units) as χ.

`IsEven(chi): GrpDrchElt -> BoolElt`

Returns `true` if and only if `Evaluate(chi,-1)` is equal to 1. Note that in characteristic 0, the space of modular forms of weight k and character χ is zero if χ is even and k is odd.

`IsOdd(chi): GrpDrchElt -> BoolElt`

Returns `true` if and only if `Evaluate(chi,-1)` is equal to -1. Note that in characteristic 0, the space of modular forms of weight k and character χ is zero if χ is odd and k is even.

`IsTotallyEven(chi): GrpDrchElt -> BoolElt`

For a Dirichlet character χ, this is `true` if and only if every character in the `Decomposition` of χ (into prime power components) is even.

`Decomposition(chi): GrpDrchElt -> List`

This decomposes the Dirichlet character χ as a product of characters with prime power moduli. The function returns a list (not a sequence) containing these characters (which do not belong to the same group).

`MinimalBaseRingCharacter(chi): GrpDrchElt -> GrpDrchElt`

This returns a character which is the same as χ, except which takes values in the smallest possible subring of the base ring of χ.

20.8.5 Evaluation

`Evaluate(chi, n): GrpDrchElt, RngIntElt -> RngElt`

`chi(n): GrpDrchElt, RngIntElt -> RngElt`

The value of the Dirichlet character χ at the integer n.

`ValueList(chi): GrpDrchElt -> [RngElt]`

A sequence containing the values $[\chi(1), .., \chi(N)]$ of the given character χ, where N is the modulus of χ.

The list of values is stored; then in later calls to `Evaluate`, the stored value is returned.

`ValuesOnUnitGenerators(chi): GrpDrchElt -> [RngElt]`

A sequence containing the values of χ on the ordered sequence of elements of $\mathbb{Z}/m$ given by `UnitGenerators(Parent(chi))`, where m is the modulus of χ.

`OrderOfRootOfUnity(r, n): RngElt, RngIntElt -> RngIntElt`

Given an element r of some ring which is *assumed* to satisfy $r^n = 1$, this returns the smallest integer m such that $r^m = 1$.

(This provides a convenient way to calculate the order of values of non-real characters.)

20.8.6 Arithmetic

`x * y: GrpDrchElt, GrpDrchElt -> GrpDrchElt`

`x / y: GrpDrchElt, GrpDrchElt -> GrpDrchElt`

The product or quotient (respectively) of the Dirichlet characters x and y. This is a Dirichlet character of modulus equal to the least common multiple of the moduli of x and y. The base rings and chosen roots of unity of the parents of x and y are equal.

`x ^ n: GrpDrchElt, RngIntElt -> GrpDrchElt`

The Dirichlet character x raised to the power of n, where n is any integer.

`x ^ phi: GrpDrchElt, Map -> GrpDrchElt`

The image of the Dirichlet character x under the automorphism ϕ.

`Sqrt(x): GrpDrchElt -> GrpDrchElt`

Given a Dirichlet character x of odd order, this returns a square root of x (in the same group).

20.8.7 Example

Example: Dirichlet Characters Modulo 5 (ex-19f11f)

We begin by constructing the group of characters $(\mathbb{Z}/5\mathbb{Z})^* \to \mathbb{Q}^*$.

```
> G<a> := DirichletGroup(5);  G;  // The default base field is Q.
Group of Dirichlet characters of modulus 5 over Rational Field
> #G;
2
> [Evaluate(a, n) : n in [1..5]];
[ 1, -1, -1, 1, 0 ]
> Eltseq(a);
[ 2 ]
> a eq G![2];
true
> IsEven(a);
true
> IsOdd(a);
false
> IsTrivial(a);
false
```

Next we create a character by building it up "locally".

```
> G1<a4> := DirichletGroup(4);
> Conductor(a4);
4
> G2<a5> := DirichletGroup(25);
> Conductor(a5);
5
> eps := a4*a5;
> Modulus(eps);
100
> Conductor(eps);
20
> Evaluate(eps,7) eq Evaluate(a4,7)*Evaluate(a5,7);
true
```

Characters can be constructed over various fields.

```
> G<a> := DirichletGroup(7,GF(7));
> #G;
6
> Evaluate(a,2);
2
>
> G<a3,a5> := DirichletGroup(15,CyclotomicField(EulerPhi(15)));
> G;
Group of Dirichlet characters of modulus 15 over Cyclotomic Field of
order 8 and degree 4
> #G;
```

(continues on next page)

(continued from previous page)

```
8
> Conductor(a3);
3
> Conductor(a5);
5
> Order(a5);
4
> Evaluate(a5,2);
zeta_8^2
```

If D is a fundamental discriminant, then `KroneckerCharacter(D)` is the quadratic Dirichlet character corresponding to the quadratic field $\mathbb{Q}(\sqrt{D})$. The following code verifies that `KroneckerCharacter` and `KroneckerSymbol` agree in the case $D = 209$.

```
> chi := KroneckerCharacter(209);
> for n in [1..209] do
>    assert Evaluate(chi,n) eq KroneckerSymbol(209,n);
> end for;
```

If E is an elliptic curve with newform f_E, then the twist E_D corresponds to f_E twisted by this character, as illustrated below.

```
> E := EllipticCurve(CremonaDatabase(),"11A");
> f := qEigenform(E,8); f;
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
> chi := KroneckerCharacter(-7);
> qEigenform(QuadraticTwist(E,-7),8);
q - 2*q^2 + q^3 + 2*q^4 - q^5 - 2*q^6 + O(q^8)
> R<q> := Parent(f);
> &+[Evaluate(chi,n)*Coefficient(f,n)*q^n : n in [1..7]] + O(q^8);
q - 2*q^2 + q^3 + 2*q^4 - q^5 - 2*q^6 + O(q^8)
```

Chapter 21

Rational Field

21.1 Introduction

This Chapter describes functions relating to the field of rational numbers $\mathbb{Q}$. Note that most functions for rational integers can be found in Chapter *Ring of Integers*.

The rational field $\mathbb{Q}$ is automatically created when Magma is started up. That means that in $\mathbb{Q}$, unlike most other structures, arithmetic can be done without the need to create the structure explicitly first. The same is true for the ring of integers.

In order to be compatible with the other rings and fields, `Q.1` will return 1.

21.1.1 Representation

Rational numbers are stored as pairs of numerator and denominator. Whenever a rational number is created, it will be put in reduced form (coprime numerator and denominator, positive denominator). It is well possible that a rational number has denominator 1, and thus represents a rational integer; in such cases it will however never automatically be converted into an integer (that is, its type will not be changed).

21.1.2 Coercion

The tables in Chapter *Introduction to Rings* describe which coercions of rational numbers are allowed, and which will take place automatically when necessary. As a general rule, automatic coercion occurs between elements of $\mathbb{Q}$ and elements of any ring R of characteristic 0. That means, for example, that addition of any rational number and an element r of such ring can be performed without the need to coerce the elements first; the result will be in the larger of $\mathbb{Q}$ and R (usually R, unless R is a subring of $\mathbb{Q}$ such as $\mathbb{Z}$). The most important exceptions to the above rule are those cases where the result would lie in a structure strictly larger than both $\mathbb{Q}$ and R. Examples of this are $R = \mathbb{Z}[x]$, and the result would generally be in $\mathbb{Q}(x)$, and $R = O_K$, an order in a number field (and the result could be in K).

Example: Automatic Coercion of Rationals (ex-77bb73)

We give three examples of successful automatic coercion, and one where it does not work. Note that in the third case the result, although being integral, is still in the rational field.

```
> 1/2 + elt< CyclotomicField(3) | 1,2>;
1/2*(4*zeta_3 + 3)
> 1/2 - 0.12345;
0.37655
> 1/2 * 2;
1
> Parent(1/2 * 2);
Rational Field
```

(continues on next page)

(continued from previous page)

```
> R<x> := PolynomialRing(Integers());
> // The following produces an error:
> 1/2 + x;

>> 1/2 + x;
       ^
Runtime error in '+': Bad argument types
```

21.1.3 Homomorphisms

Since homomorphisms are generally only allowed to be unitary, the specification of ring homomorphisms from $Q = \mathbb{Q}$ to a ring R is particularly simple: the image is completely determined by the image of 1, which we require to be 1 in R, so

hom< Q -> R | >: FldHom, Rng -> Map

suffices.

Note that Magma allows the user to define maps with **hom** that are not proper homomorphisms; this is sometimes useful, as the example below shows.

Example: Mapping Rationals into the Integers Modulo 11 (ex-d98da9)

Suppose we wish to coerce rational numbers with denominator not divisible by 11 into the ring $\mathbb{Z}/11\mathbb{Z}$ in the obvious way by sending r/s to rs^{-1} mod 11. The coercion rules do not allow you to do so using **!**, but a simple 'homomorphism' will work.

```
> Z11 := Integers(11);
> Q := RationalField();
> h := hom< Q -> Z11 | >;
> h(1/2);
6
```

21.2 Creation Functions

21.2.1 Creation of Structures

The rational field $\mathbb{Q}$ is automatically created when Magma is started up. Nevertheless, it may be necessary to formally create the rational field, for instance if it is to be used as the coefficient ring for a polynomial ring. There is a unique rational field structure in Magma, that is, multiple calls to the creation function **RationalField()** will return the same object (and not an isomorphic copy), so no memory will be wasted.

Rationals() -> FldRat

RationalField() -> FldRat

Create the field $\mathbb{Q}$ of rational numbers.

MaximalOrder(Q): FldRat -> RngInt

IntegerRing(Q): FldRat -> RngInt

IntegerRing() -> RngInt

Integers() -> RngInt

RingOfIntegers(Q): FldRat -> RngInt

Create the field $\mathbb{Z}$ of rational integers.

FieldOfFractions(Q): FldRat -> FldRat

FieldOfFractions(Z): RngInt -> FldRat

The function **FieldOfFractions** returns the field $\mathbb{Q}$ when R is either the ring $\mathbb{Z}$ of rational integers, or the field $\mathbb{Q}$ itself.

`Completion(Q, P): FldRat, RngInt -> FldLoc, Map`

`Precision: RngIntElt`	`Default: Infinity()`

Computes the completion of Q at the integral prime ideal P together with the injection into the completion.

The parameter `Precision` may be used to specify a particular precision.

21.2.2 Creation of Elements

Unlike elements of other structures, rational numbers and integers can be created as literals without the need to define the parent field $\mathbb{Q}$ or the parent ring $\mathbb{Z}$ first, since these structures are loaded whenever Magma is started up.

`a / b: RngIntElt, RngIntElt -> FldRatElt`

Given integers a and $b \neq 0$, form the rational number a/b (in reduced form). Of course a and b are allowed to be given as expressions defining integers.

`Q ! [a]: FldRat, RngElt -> FldRatElt`

The inverse function to *Eltseq*, returns `Q!a`.

`Q ! [a, b]: FldRat, RngIntElt, RngIntElt -> FldRatElt`

`elt< Q | a, b >: FldRat, RngIntElt, RngIntElt -> FldRatElt`

Given the rational field $\mathbb{Q}$, and integers a, b (with $b \neq 0$), construct the rational number a/b, in reduced form.

`Q ! a: FldRat, RngIntElt -> FldRatElt`

Given the rational field $\mathbb{Q}$, and an integer a, create the rational number $a = a/1$ in $\mathbb{Q}$. Also, any element from a quadratic, cyclotomic, or number field (or an order of such) that is rational can be coerced into the rational field this way.

`One(Q): FldRat -> FldRatElt`

`Identity(Q): FldRat -> FldRatElt`

`Zero(Q): FldRat -> FldRatElt`

`Representative(Q): FldRat -> FldRatElt`

These generic functions (cf. Chapter *Introduction to Rings*) create 1, 1, 0, and 0 respectively, in the rational field $\mathbb{Q}$.

`RootOfUnity(n, Q): RngIntElt, FldRat -> FldRatElt`

This function returns, in general, for a positive integer n and a cyclotomic field Q a primitive n-th root of unity in Q; if Q is the rational field, n must be 1 or 2, and the result will be 1 or -1 in Q accordingly.

`Random(Q, m): FldRat, RngIntElt -> FldRatElt`

This function returns a random rational number with random numerator in $[-u..u]$ and random denominator in $[1..u]$, where u is the absolute value of m.

21.3 Structure Operations

21.3.1 Related Structures

`Category(Q): FldRat -> Cat`

`Parent(Q): FldRat -> PowerStructure`

`PrimeField(Q): FldRat -> FldRat`

`IntegralBasis(Q): FldRat -> [ FldRatElt ]`

An integral basis for Q as a number field as a sequence of elements of Q (giving the sequence containing 1 for the rational field).

`MinimalField(q): FldRatElt -> FldRat`

Return the least cyclotomic field containing the cyclotomic field element q; if q is rational this returns the rational field.

`MinimalField(S): SetEnum -> FldRat`

Returns the minimal cyclotomic field containing the cyclotomic field elements in the enumerated set S; this will return the rational field if all elements of S are rational numbers.

`BaseField(Q): FldRat -> FldRat`

In analogy to the number fields, returns the coefficient field of Q which will be $\mathbb{Q}$.

`Basis(Q): FldRat -> [FldRatElt]`

`AbsoluteBasis(Q): FldRat -> [FldRatElt]`

A basis for Q as a $\mathbb{Q}$-vector space, i.e. [1].

`UnitGroup(Q): FldRat -> GrpAb, Map`

The unit group of the maximal order of $\mathbb{Q}$ (i.e. of $\mathbb{Z}$).

`ClassGroup(Q): FldRat -> GrpAb, Map`

The class group of the ring of integers $\mathbb{Z}$ of $\mathbb{Q}$ (which is trivial).

`AutomorphismGroup(Q): FldRat -> GrpPerm, PowMapAut, Map`

`AutomorphismGroup(Q, Q): FldRat, FldRat -> GrpPerm, PowMapAut, Map`

The group of $\mathbb{Q}$ automorphisms of $\mathbb{Q}$, ie. a trivial finitely presented group, the parent structure for $\mathbb{Q}$-automorphisms and a map from the group to actual field automorphisms. In this case, of course the only $\mathbb{Q}$-automorphism will be the identity.

`Algebra(Q, Q): FldRat, Fld -> AlgAss, Map`

The rational numbers form canonically an algebra. This function returns an associative $\mathbb{Q}$-algebra isomorphic to $\mathbb{Q}$ and the map from the algebra to $\mathbb{Q}$.

`VectorSpace(Q, Q): FldRat, Fld -> ModTupFld, Map`

The rational numbers form canonically a vector space. This function returns a $\mathbb{Q}$-vector space isomorphic to $\mathbb{Q}$ and the map from the vector space to $\mathbb{Q}$.

`Decomposition(Q, p): FldRat, RngIntElt -> []`

`Decomposition(Q, p): FldRat, Infty -> []`

For a prime p or for the "infinite prime" `Infinity()` compute the decomposition in $\mathbb{Q}$ as a number field. This returns a list of length one containing a 2-tuple describing the splitting behaviour: the first component contains p and the second it's ramification degree, ie. 1.

21.3.2 Numerical Invariants

The functions below are defined for the rational field $\mathbb{Q}$ mainly because it often arises as a degenerate case of quadratic or cyclotomic field constructions. See the corresponding Chapters Quadratic Fields and Cyclotomic Fields for more.

`Characteristic(Q): FldRat -> RngIntElt`

`Conductor(Q): FldRat -> RngIntElt`

The smallest positive integer n such that Q is contained in the cyclotomic field $\mathbb{Q}(\zeta_n)$. For the rational field this is 1.

`Degree(Q): FldRat -> RngIntElt`

`AbsoluteDegree(Q): FldRat -> RngIntElt`

The degree of Q as a number field (which is 1 for the rational field).

`Discriminant(Q): FldRat -> RngIntElt`

`AbsoluteDiscriminant(Q): FldRat -> RngIntElt`

The field discriminant of Q (which is 1 for the rational field).

DefiningPolynomial(Q): FldRat -> RngUPolElt

An irreducible polynomial over $\mathbb{Q}$ a root of which generates Q as a number field (for the rational field this returns the linear polynomial $x - 1$).

Signature(Q): FldRat -> RngIntElt, RngIntElt

The signature (number of real embeddings and pairs of complex embeddings) of $\mathbb{Q}$.

21.3.3 Ring Predicates and Booleans

IsCommutative(Q): FldRat -> BoolElt

IsUnitary(Q): FldRat -> BoolElt

IsFinite(Q): FldRat -> BoolElt

IsOrdered(Q): FldRat -> BoolElt

IsField(Q): FldRat -> BoolElt

IsEuclideanDomain(Q): FldRat -> BoolElt

IsPID(Q): FldRat -> BoolElt

IsUFD(Q): FldRat -> BoolElt

IsDivisionRing(Q): FldRat -> BoolElt

IsEuclideanRing(Q): FldRat -> BoolElt

IsPrincipalIdealRing(Q): FldRat -> BoolElt

IsDomain(Q): FldRat -> BoolElt

Q **eq** R: FldRat, FldRat -> BoolElt

Q **eq** R: FldRat, RngInt -> BoolElt

Q **ne** R: FldRat, FldRat -> BoolElt

Q **ne** R: FldRat, RngInt -> BoolElt

21.4 Element Operations

A variety of different types of operations are provided for rational elements including arithmetic operations, comparison and predicates, and converting to a sequence.

21.4.1 Parent and Category

Parent(r): FldRatElt -> FldRat

Category(r): FldRatElt -> Cat

21.4.2 Arithmetic Operators

+ a: FldRatElt -> FldRatElt

- a: FldRatElt -> FldRatElt

a **+** b: FldRatElt, FldRatElt -> FldRatElt

a **-** b: FldRatElt, FldRatElt -> FldRatElt

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

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

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

`a +:= b: FldRatElt, FldRatElt -> FldRatElt`

`a -:= b: FldRatElt, FldRatElt -> FldRatElt`

`a *:= b: FldRatElt, FldRatElt -> FldRatElt`

`a /:= b: FldRatElt, FldRatElt -> FldRatElt`

`a ^:= k: FldRatElt, RngIntElt -> FldRatElt`

21.4.3 Numerator and Denominator

`Numerator(q): FldRatElt -> RngIntElt`

The (integer) numerator of the rational number q in reduced form.

`Denominator(q): FldRatElt -> RngIntElt`

The (integer) denominator of the rational number q in reduced form. This will always be a positive integer.

Example: Numerator and Denominator (ex-da083f)

Rational numbers are always immediately put in reduced form, that is, the greatest common divisor of numerator and denominator is taken out, and the denominator will be positive.

```
> Numerator(10/-4);
-5
> Denominator(10/-4);
2
```

21.4.4 Equality and Membership

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

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

`a in R: FldRatElt, Rng -> BoolElt`

`a notin R: FldRatElt, Rng -> BoolElt`

21.4.5 Predicates on Ring Elements

`IsIntegral(q): FldRatElt -> BoolElt`

Returns `true` if the rational number q is an element of the ring of integers, `false` otherwise.

`IsZero(a): FldRatElt -> BoolElt`

`IsOne(a): FldRatElt -> BoolElt`

`IsMinusOne(a): FldRatElt -> BoolElt`

`IsNilpotent(a): FldRatElt -> BoolElt`

`IsIdempotent(a): FldRatElt -> BoolElt`

`IsUnit(a): FldRatElt -> BoolElt`

`IsZeroDivisor(a): FldRatElt -> BoolElt`

`IsRegular(a): FldRatElt -> BoolElt`

`IsIrreducible(a): FldRatElt -> BoolElt`

`IsPrime(a): FldRatElt -> BoolElt`

21.4.6 Comparison

`a gt b: FldRatElt, FldRatElt -> BoolElt`

`a ge b: FldRatElt, FldRatElt -> BoolElt`

`a lt b: FldRatElt, FldRatElt -> BoolElt`

`a le b: FldRatElt, FldRatElt -> BoolElt`

`Maximum(a, b): FldRatElt, FldRatElt -> FldRatElt`

`Maximum(Q): [FldRatElt] -> FldRatElt`

`Minimum(a, b): FldRatElt, FldRatElt -> FldRatElt`

`Minimum(Q): [FldRatElt] -> FldRatElt`

21.4.7 Conjugates, Norm, and Trace

`ComplexConjugate(q): FldRatElt -> FldRatElt`

The complex conjugate of q, which will be the rational number q itself.

`Conjugate(q): FldRatElt -> FldRatElt`

The conjugate of q, which will be the rational number q itself.

`Norm(q): FldRatElt -> FldRatElt`

The norm (in $\mathbb{Q}$) of q, which will be the rational number q itself.

`Trace(q): FldRatElt -> FldRatElt`

The trace (in $\mathbb{Q}$) of q, which will be the rational number q itself.

`MinimalPolynomial(q): FldRatElt -> RngUPolElt`

Returns the minimal polynomial of the rational number q, which is the monic linear polynomial with constant coefficient q in a univariate polynomial ring R over the rational field. (If R has not been created before with a name for its indeterminate, `$.1-q` will be returned.)

21.4.8 Absolute Value and Sign

`AbsoluteValue(q): FldRatElt -> FldRatElt`

`Abs(q): FldRatElt -> FldRatElt`

The absolute value $|q|$ of a rational number q.

`Sign(q): FldRatElt -> RngIntElt`

Returns the sign of the rational number q, which is one of the integers -1, 0, 1, corresponding to the cases $q < 0$, $q = 0$, and $q > 0$.

`Height(q): FldRatElt -> RngIntElt`

The height of $q = r/s$. For r and s coprime, the height is defined as the maximum of the absolute value of r and s.

21.4.9 Rounding and Truncating

`Ceiling(q): FldRatElt -> RngIntElt`

The ceiling of the rational number q, that is, the least integer greater than or equal to q.

`Floor(q): FldRatElt -> RngIntElt`

The floor of the rational number q, that is, the largest integer less than or equal to q.

`Round(q): FldRatElt -> RngIntElt`

This function returns the integer value of the rational number q rounded to the nearest integer. In the case of a tie, rounding is done away from zero (that is, $i+\frac{1}{2}$ is rounded to $i+1$, for non-negative integers i and $i-\frac{1}{2}$ is rounded to $i-1$, for non-positive integers i).

`Truncate(q): FldRatElt -> RngIntElt`

This function returns the integer truncation of the rational number q, that is the integral part of q. Thus the effect is that of rounding towards 0.

`Qround(q, M): FldRatElt, RngIntElt -> FldRatElt`

`ContFrac: BoolElt`	`Default: false`

Finds a rational approximation d of q such that the denominator of d is bounded by M. If `ContFrac` is given then an optimal approximation is computed using the continued fraction process. By default d is obtained by some rounding procedure which is faster but gives worse results.

21.4.10 Continued Fractions

`ContinuedFraction(r): FldRatElt -> [ RngIntElt ]`

Given a rational r, return the sequence of partial quotients of the continued fraction expansion of r.

`ContinuedFractionValue(C): [ RngIntElt ] -> FldRatElt`

Given a continued fraction expansion C, return the rational r such that C equals `ContinuedFraction(r)`.

`HirzebruchJungContinuedFraction(r): FldRatElt -> [ RngIntElt ]`

`HJContinuedFraction(r): FldRatElt -> [ RngIntElt ]`

Given a rational r, return the sequence of partial quotients of the Hirzebruch-Jung continued fraction expansion of r.

`HirzebruchJungContinuedFractionValue(C): [ RngIntElt ] -> FldRatElt`

`HJContinuedFractionValue(C): [ RngIntElt ] -> FldRatElt`

Given a Hirzebruch-Jung continued fraction expansion C, return the rational r so that `HJContinuedFraction(r)` equals C.

21.4.11 Rational Reconstruction

Under certain circumstances it is useful to have a partial inverse of the function $\psi_m\colon \mathbb{Q} \to \mathbb{Z}/m\mathbb{Z}$ of taking residues modulo m (where the obvious value of ψ_m is only defined for rational numbers with denominator in smallest terms coprime to m); the partial inverse of the function is sometimes referred to as 'rational reconstruction'. For $s \in \mathbb{Z}/m\mathbb{Z}$ the value of $\psi^{-1}(s)$ is the rational number r for which $\psi_m(r) = s$ and, in addition, the absolute values of both the numerator and denominator of r are at most $\sqrt{m/2}$; such r does not always exist, but if r exists it is unique.

`RationalReconstruction(s): RngIntResElt -> BoolElt, FldRatElt`

`RationalReconstruction(s): FldFinElt -> BoolElt, FldRatElt`

Given an element s of a ring S of m elements, return a Boolean flag indicating whether or not a rational number r exists such that for the representation $r = n/d$ in minimal terms it holds that $n \cdot d^{-1} \equiv s \bmod m$, $|n| \leq \sqrt{m/2}$ and $0 < d \leq \sqrt{m/2}$. If the flag is true, the element r is also

returned. The ring S is allowed to be a residue class ring `Integers(m)` or a finite field of prime cardinality $p = m$: `FiniteField(p)`.

In addition, s is allowed to be a matrix over a prime finite field, in which case the existence (and, if possible, value) of a rational reconstruction of the matrix is determined.

21.4.12 Valuation

`Valuation(x, p): FldRatElt, RngIntElt -> RngIntElt, FldRatElt`

`Valuation(x, I): FldRatElt, RngIntElt -> RngIntElt, FldRatElt`

The valuation v of the rational number x at the prime p (the prime ideal I). This is the difference of the valuations of the numerator and denominator of x. The optional second return value is the rational u such that $x = p^v u$.

21.4.13 Sequence Conversions

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

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

The sequence $[a]$ for compatibility with the other field types.

Chapter 22

Finite Fields

22.1 Introduction

Magma provides a powerful environment for computing with lattices of finite fields. Complete freedom in the manner in which fields are constructed is allowed, while assuring compatibility. Finite fields of various kinds are supported, with optimized representations for each kind. For a detailed description of how finite fields are presented in Magma, see [BCS97].

22.1.1 Representation of Finite Fields

In Magma, arithmetic in small non-prime finite fields is carried out using tables of Zech logarithms. While this ensures that finite field arithmetic is fast, its use is limited to finite fields of small cardinality. Larger finite fields are internally represented as polynomial rings over a small finite field. It is possible for the user to specify his own irreducible polynomial (although internally an alternative representation may well be used). Although two finite fields of the same cardinality are isomorphic, in practical applications it is often important to be guaranteed to work in a field defined by a specific polynomial. Moreover, in passing between fields and subfields, choices regarding the embeddings have to be made, so that these embeddings are *compatible* (so that 'diagrams commute'). The scheme implemented in Magma and described in [BCS97] ensures that this is so.

22.1.2 Conway Polynomials

To avoid ambiguities when talking about (small) finite fields, *Conway polynomials* have been defined and calculated by R. Parker. The Conway polynomial $C_{p,n}$ is the lexicographically first monic irreducible, primitive polynomial of degree n over $\mathbf{F}_p$ with the property that it is consistent with all $C_{p,m}$ for m dividing n. Consistency of $C_{p,n}$ and $C_{p,m}$ for m dividing n means that for a root α of $C_{p,n}$ it holds that $\beta = \alpha^{\frac{p^n-1}{p^m-1}}$ is a root of $C_{p,m}$. Lexicographically first is with respect to the system of representatives $-\frac{p-1}{2}, \ldots, -1, 0, 1, \ldots, \frac{p-1}{2}$ for the residue classes modulo p, ordered via $0 < -1 < 1 < -2 < \cdots \frac{p-1}{2}$ (and we only need to compare polynomials of the same degree). To compute the Conway polynomial $C_{p,n}$ one needs to know all Conway polynomials $C_{p,m}$ for m dividing n, and as far as we know, no essentially better method is known than enumerating and testing the primitive polynomials of degree n in lexicographical order. Conway polynomials are used in Magma by default for the construction of $\mathbf{F}_{p^n}$ using `FiniteField(p, n)` or its synonyms, whenever the Conway polynomial is available. However, it must be stressed that Conway polynomials are **only** used in Magma to provide standard defining polynomials for $\mathbf{F}_{p^n}$ – the special properties of Conway polynomials are never used because they are totally irrelevant in the scheme implemented in Magma!

22.1.3 Ground Field and Relationships

Throughout this Chapter we will use the notions of *ground field* and *prime field* in the following way. The prime field of a finite field F is the unique field of cardinality p, the characteristic of F. Here we mean unique not just in the mathematical sense, but in the sense that all prime fields of the same cardinality are identical in Magma, and their elements are denoted $0, 1, \ldots, p-1$. The ground field of F is the field

over which F is created as an extension. If F was not explicitly created as an extension of a finite field E by using `ext`, its ground field will be the prime field. Printing in Magma always takes place with respect to the ground field, that is, elements of F are expressed as polynomials in the generator `F.1` of the field with coefficients in the ground field; there is one exception to this rule: if the field F is small enough to be represented by means of Zech logarithms, the printing of elements is in the form of powers of the primitive element (see the option on `AssertAttribute` below for ways of changing that). It should be kept in mind that finite fields may be related mathematically without Magma being aware of the relation between them. This happens for example when two fields of dividing degrees are created as *extensions* of one field; although an isomorphic image of the smaller field will be contained in the larger, Magma will not establish this relation (unless the user explicitly asks for it, using the `Embed` function). However, all `subfields` of one common overfield in Magma will have their inclusion relations set up automatically.

22.2 Creation Functions

A database of low-term irreducible polynomials for finite fields of small characteristic is available in Magma (see the function *IrreducibleLowTermGF2Polynomial* below). This database goes up to degree 120,000 for $\mathbf{F}_2$, and to reasonably high degrees for the fields $\mathbf{F}_q$ for q up to 128. Thus one can create the finite field $\mathbf{F}_{p^k}$ for p, k within this range and compute within the field without any delay in the creation. Advantage is also taken of the special form of the defining polynomial. Previous to V2.11, sparse trinomial/pentanomial irreducible polynomials (see the function *IrreducibleSparseGF2Polynomial*) were used by default for constructing $GF(2^k)$ when k is beyond the Conway range. To enable compatibility with older versions, one may select these sparse polynomials with the parameter `Sparse` in the creation functions.

22.2.1 Creation of Structures

`FiniteField(q): RngIntElt -> FldFin`

`GaloisField(q): RngIntElt -> FldFin`

`GF(q): RngIntElt -> FldFin`

```
Optimize: BoolElt                    Default: true
Sparse  : BoolElt                    Default: false
```

Given $q = p^n$, where p is a prime, create the finite field $\mathbf{F}_q$. If p is very big, it is advised to use the form `FiniteField(p, n)` described below instead, because Magma will first attempt to factor q completely. The primitive polynomial used to construct $\mathbf{F}_q$ when $n > 1$ will be a Conway polynomial, if it is available. If the parameter `Optimize` is `false`, then no optimized representation (i.e., by using Zech logarithm tables or internal multi-step extensions) will be constructed for the new field which means that the time to create the field will be trivial but arithmetic operations in the field may be slower – this is useful if say one wishes to just compute a few trivial operations on a few elements of the field alone. If $q = 2^k$ and k is beyond the Conway range, then a low-term irreducible is used (see *IrreducibleLowTermGF2Polynomial* below). Setting the parameter `Sparse` to `true` will cause a sparse polynomial to be used instead if possible (see *IrreducibleSparseGF2Polynomial* below).

`FiniteField(p, n): RngIntElt, RngIntElt -> FldFin`

`GaloisField(p, n): RngIntElt, RngIntElt -> FldFin`

`GF(p, n): RngIntElt, RngIntElt -> FldFin`

```
Check   : BoolElt                    Default: true
Optimize: BoolElt                    Default: true
Sparse  : BoolElt                    Default: false
```

Given a prime p and an exponent $n \geq 1$, create the finite field $\mathbf{F}_{p^n}$. The primitive polynomial used to construct $\mathbf{F}_q$ when $n > 1$ will be a Conway polynomial, if it is available. By default p is checked to be a strong pseudoprime for 20 random bases b with $1 < b < p$; if the parameter `Check`

is `false`, then no check is done on p at all (this is useful when p is very large and one does not wish to perform an expensive primality test on p). The parameters are as above.

`ext<F | n>: FldFin, RngIntElt -> FldFin, Map`

```
Optimize: BoolElt                    Default: true
Sparse  : BoolElt                    Default: false
```

Given a finite field F and a positive integer n, create an extension G of degree n of F, as well as the embedding map $\phi : F \to G$. The parameter `Optimize` has the same behaviour as that for the `FiniteField` function. If F is a default field, then G will also be a default field (so its ground field will be the prime field). Otherwise, the ground field of G will be F. The parameters are as above.

`ext<F | P>: FldFin, RngUPolElt[FldFin] -> FldFin, Map`

```
Optimize: BoolElt                    Default: true
```

Given a finite field F and a polynomial P of degree n over F, create an extension $G = F[\alpha]$ of degree n of F, as well as the natural embedding map $\phi : F \to G$; the polynomial P must be irreducible over F, and α is one of its roots. Thus the defining polynomial of G over F will be P. The parameter `Optimize` has the same behaviour as that for the `FiniteField` function. The ground field of G will be F. The parameter is as above.

`ExtensionField<F, x | P>: FldFin, ... -> FldFin, Map`

Given a finite field F, a literal identifier x, and a polynomial P of degree n over F presented as a (polynomial) expression in x, create an extension $G = F[x]$ of degree n of F, as well as the natural embedding map $\phi : F \to G$; the polynomial P must be irreducible over F, and x is one of its roots. Thus the defining polynomial of G over F will be P. The parameter `Optimize` has the same behaviour as in the `FiniteField` function.

`RandomExtension(F, n): FldFin, RngIntElt -> FldFin`

Given a finite field F and a degree n, return the extension of F by a random degree-n irreducible polynomial over F.

`SplittingField(P): RngUPolElt[FldFin] -> FldFin`

Given a univariate polynomial P over a finite field F, create the minimal splitting field of P, that is, the smallest-degree extension field G of F such that P factors completely into linear factors over G.

`SplittingField(S): { RngUPolElt[FldFin] } -> FldFin`

Given a set S of univariate polynomials each over a finite field F, create the minimal splitting field of S, that is, the smallest-degree extension field G of F such that for every polynomial P of S, P factors completely into linear factors over G.

`sub<F | d>: FldFin, RngIntElt -> FldFin, Map`

```
Optimize: BoolElt                    Default: true
Sparse  : BoolElt                    Default: false
```

Given a finite field F of cardinality p^n and a positive divisor d of n, create a subfield E of F of degree d, as well as the embedding map $\phi : E \to F$. The parameters are as above.

`sub<F | f>: FldFin, FldFinElt -> FldFin, Map`

```
Optimize: BoolElt                    Default: true
Sparse  : BoolElt                    Default: false
```

Given a finite field F and an element f of F, create the subfield E of F generated by f, together with the embedding map $\phi : E \to F$. The map and field are constructed so that $\phi(w) = f$, where w is the generator of E (that is, `E.1`). The parameters are as above.

`GroundField(F): FldFin -> FldFin`

`BaseField(F): FldFin -> FldFin`

Given a finite field F, return its ground field. If F was constructed as an extension of the field E, this function returns E; if F was not explicitly constructed as an extension then the prime field is returned.

`PrimeField(F): FldFin -> FldFin`

The subfield of F of prime cardinality.

`IsPrimeField(F): Fld -> BoolElt`

Returns whether field F is a prime field.

`F meet G: FldFin, FldFin -> FldFin`

Given finite fields F and G of the same characteristic p, return the finite field that forms the intersection $F \cap G$.

`CommonOverfield(K, L): FldFin, FldFin -> FldFin`

Given finite fields K and L, both of characteristic p, return the smallest field which contains both of them.

Example: Ways to Define a Finite Field (ex-591238)

To define the field of 7 elements, use

```
> F7 := FiniteField(7);
```

We can define the field of 7^4 elements in several different ways. We can use the Conway polynomial:

```
> F<z> := FiniteField(7^4);
> F;
Finite field of size 7^4
```

We can define it as an extension of the field of 7 elements, using the internal polynomial:

```
> F<z> := ext< F7 | 4 >;
> F;
Finite field of size 7^4
```

We can supply our own polynomial, say $x^4 + 4x^3 + 2x + 3$:

```
> P<x> := PolynomialRing(F7);
> p := x^4+4*x^3+2*x+3;
> F<z> := ext< F7 | p >;
> F;
Finite field of size 7^4
```

We can define it as an extension of the field of 7^2 elements:

```
> F49<w> := ext< F7 | 2 >;
> F<z> := ext< F49 | 2 >;
> F;
Finite field of size 7^4
```

22.2.2 Creating Relations

`Embed(E, F): FldFin, FldFin`

Given finite fields E and F of cardinality p^d and p^n, such that d divides n, assert the embedding relation between E and F. That is, an isomorphism between E and the subfield of F of cardinality p^d is chosen and set up, and can be used from then on to move between the fields E and F. See [BCS97] for details as to how this is done. If both E and F have been defined with Conway

polynomials then the isomorphism will be such that the generator β of F is mapped to $\alpha^{\frac{p^n-1}{p^d-1}}$, where α is the generator of F.

`Embed(E, F, x): FldFin, FldFin, FldFinElt`

Given finite fields E and F of cardinality p^d and p^n such that d divides n, as well as an element $x \in F$, assert the embedding relation between E and F mapping the generator of E to x. The element x must be a root of the polynomial defining E over the prime field. Thus an isomorphism between E and the subfield of F of cardinality p^d is set up, and can be used from then on to move between the fields E and F.

`IsIsomorphic(E, F): FldFin, FldFin -> BoolElt, Map[FldFin, FldFin]`

Given finite fields E and F, returns whether they are isomorphic (have the same cardinality), and returns an isomorphism $\phi : E \to F$, without implicitly embedding E into F.

22.2.3 Special Options

For finite fields for which the complete table of Zech logarithms is stored (and which must therefore be small), printing of elements can be done in two ways: either as powers of the primitive element or as polynomials in the generating element. Note that power printing is not available in all cases where the logarithm table is stored however (the defining polynomial may not be primitive); for convenience element of a prime field are always printed as integers, and therefore power printing on prime fields will not work. Also, if a field is created with a generator that is not primitive, then power printing will be impossible.

`AssertAttribute(FldFin, "PowerPrinting", l): Cat, MonStgElt, BoolElt`

This attribute is used to change the *default* printing for all (small) finite fields created after the `AssertAttribute` command is executed. If l is `true` all elements of finite fields small enough for the Zech logarithms to be stored will be printed by default as a power of the primitive element – see `PrimitiveElement`). If l is `false` every finite field element is printed by default as a polynomial in the generator `F.1` of degree less than n over the ground field. The default can be overruled for a particular finite field by use of the `AssertAttribute` option listed below. The value of this attribute is obtained by use of `HasAttribute(FldFin, "PowerPrinting")`.

`SetPowerPrinting(F, l): FldFin, BoolElt`

`AssertAttribute(F, "PowerPrinting", l): FldFin, MonStgElt, BoolElt`

Given a finite field F, the Boolean value l can be used to control the printing of elements of F, provided that F is small enough for the table of Zech logarithms to be stored. If l is `true` all elements will be printed as a power of the primitive element – see `PrimitiveElement`). If l is `false` (which is the only possibility for big fields), every element of F is printed as a polynomial in the generator `F.1` of degree less than n over the ground field of F, where n is the degree of F over its ground field. The function `HasAttribute(F, "PowerPrinting")` may be used to obtain the current value of this flag.

`HasAttribute(FldFin, "PowerPrinting", l): Cat, MonStgElt, BoolElt`

This function is used to find the current default printing style for all (small) finite fields. It returns `true` (since this attribute is always defined for FldFin), and also returns the current value of the attribute. The procedure `AssertAttribute(FldFin, "PowerPrinting", l)` may be used to control the value of this flag.

`HasAttribute(F, "PowerPrinting"): FldFin, MonStgElt -> BoolElt, BoolElt`

Given a finite field F that is small enough for the table of Zech logarithms to be stored, returns `true` if the attribute `"PowerPrinting"` is defined, else returns `false`. If the attribute is defined, the function also returns the value of the attribute. The procedure `AssertAttribute(F, "PowerPrinting", l)` may be used to control the value of this flag.

`AssignNames(~F, [f]): FldFin, [ MonStgElt ]`

Procedure to change the name of the generating element in the finite field F to the contents of the string f. When F is created, the name will be `F.1`. This procedure only changes the name used in printing the elements of F. It does *not* assign to an identifier called f the value of the generator 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, 1): FldFin, RngIntElt -> FldFinElt`

Given a finite field F, return the element which has the name attached to it, that is, return the element `F.1` of F.

22.2.4 Homomorphisms

`hom< F -> G | x >: FldFin, Rng -> Map`

Given a finite field F, create a homomorphism with F as its domain and G as its codomain. If F is a prime field, then the right hand side in the constructor must be empty; in this case the ring homomorphism is completely determined by the rule that the map must be unitary, that is, 1 of F is mapped to 1 of G. If F is not of prime cardinality, then the homomorphism must be specified by supplying one element x in the codomain, which serves as the image of the generator of the field F over its prime field. Note that it is the responsibility of the user that the map defines a homomorphism.

22.2.5 Creation of Elements

`F . 1: FldFin -> FldFinElt`

The generator for F as an algebra over its ground field. Thus, if F was defined by the polynomial $P = P(X)$ over E, so $F \cong E[X]/P(X)$, then `F.1` is the image of X in F. If F is a prime field, then $1 = 1_F$ will be returned.

`elt<F | a>: FldFin, RngElt -> FldFinElt`

`F ! a: FldFin, RngElt -> FldFinElt`

Given a finite field F create the element 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 a subfield of F;

(iii) an element of an overfield of F that lies in F;

(iv) an integer, to be identified with a modulo the characteristic of F;

(v) a sequence of elements of the ground field E of F, of length equal to the degree of F over E. In this case the element $a_0 + a_1 w + \cdots + a_{n-1} w^{n-1}$ is created, where $a = [a_0, \ldots a_{n-1}]$ and w is the generator `F.1` of F over E.

`elt<F | a0, ..., an-1>: FldFin, [FldFinElt] -> FldFinElt`

Given a finite field F with generator w of degree n over the ground field E, create the element $a_0 + a_1 w + \cdots + a_{n-1} w^{n-1} \in F$, where $a_i \in E$ $(0 \leq i \leq n-1)$. If the a_i are in some subfield of E or the a_i are integers, they will be coerced into the ground field.

`One(F): FldFin -> FldFinElt`

`Identity(F): FldFin -> FldFinElt`

`Zero(F): FldFin -> FldFinElt`

`Representative(F): FldFin -> FldFinElt`

These generic functions (cf. Chapter *Introduction to Rings*) create 1, 1, 0, and 0 respectively, in any finite field.

`Random(F): FldFin -> FldFinElt`

Create a 'random' element of finite field F.

22.2.6 Special Elements

`F . 1: FldFin, RngIntElt -> FldFinElt`

`Generator(F): FldFin -> FldFinElt`

Given a finite field F, this function returns the element f of F that generates F over its ground field E, so $F = E[f]$. This is the same as the element `F.1`.

`Generator(F, E): FldFin, FldFin -> FldFinElt`

Given a finite field F and a subfield E of F, this function returns an element f of F that generates F over E, so $F = E[f]$. Note that this element may be different from the element `F.1`, but if `F.1` works it will be returned.

`PrimitiveElement(F): FldFin -> FldFinElt`

Given a finite field F, this function returns a primitive element for F, that is, a generator for the multiplicative group F^* of F. Note that this may be an element different from the generator `F.1` for the field as an algebra. This function will return the same element upon different calls with the same field; the primitive element that is returned is the one that is used as basis for the `Log` function.

`SetPrimitiveElement(F, x): FldFin, FldFinElt`

(Procedure.) Given a finite field F and a *primitive* element x of F, set the internal primitive element p of F to be x. If the internal primitive element p of F has already been computed or set, x must equal it. The function `Log` (given one argument) returns the logarithm of its argument with respect to the base p; this function thus allows one to specify which base should be used. (One can also use `Log(x, b)` for a given base but setting the primitive element and using `Log(x)` will be faster if many logarithms are to be computed.)

`NormalElement(F): FldFin -> FldFinElt`

Given a finite field $F = \mathbf{F}_{p^n}$, this function returns a normal element for F over the ground field G, that is, an element $\alpha \in F$ such that $\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}$ forms a basis for F over G, where q is the cardinality of G, and n the degree for F over G. Two calls to this function with the same field may result in different normal elements.

`NormalElement(F, E): FldFin, FldFin -> FldFinElt`

Given a finite field $F = \mathbf{F}_{q^n}$ and a subfield $E = \mathbf{F}_q$, this function returns a normal element for F over E, that is, an element $\alpha \in F$ such that $\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}$ forms a basis for F over E.

22.2.7 Sequence Conversions

`SequenceToElement(s, F): [ FldFinElt ] -> FldFinElt`

`Seqelt(s, F): [ FldFinElt ] -> FldFinElt`

Given a sequence $s = [s_0, \ldots, s_{n-1}]$ of elements of a finite field E, of length equal to the degree of the field F over its subfield E, construct the element $s = s_0 + s_1 w + \cdots + s_{n-1} w^{n-1}$ of F, where w is the generator `F.1` of F over E.

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

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

Given an element a of the finite field F, return the sequence of coefficients $[a_0, \ldots, a_{n-1}]$ in the ground field E of F such that $a = a_0 + a_1 w + \cdots + a_{n-1} w^{n-1}$, with w the generator of F over E, and n the degree of F over E.

`ElementToSequence(a, E): FldFinElt, FldFin -> [ FldFinElt ]`

`Eltseq(a, E): FldFinElt, FldFin -> [ FldFinElt ]`

Given an element a of the finite field F, return the sequence of coefficients $[a_0, \ldots, a_{n-1}]$ in the subfield E of F such that $a = a_0 + a_1 w + \cdots + a_{n-1} w^{n-1}$, with w the generator of F over E, and n the degree of F over E.

22.3 Structure Operations

22.3.1 Related Structures

`Category(F): FldFin -> Cat`

`Parent(F): FldFin -> PowerStructure`

`Centre(F): FldFin -> FldFin`

`PrimeRing(F): FldFin -> FldFin`

`PrimeField(F): FldFin -> FldFin`

`FieldOfFractions(F): FldFin -> FldFin`

`AdditiveGroup(F): FldFin -> GrpAb, Map`

Given $F = \mathbf{F}_q$, create the finite additive abelian group A of order $q = p^r$ that is the direct sum of r copies of the cyclic group of order p, together with the corresponding isomorphism from the group A to the field F.

`MultiplicativeGroup(F): FldFin -> GrpAb, Map`

`UnitGroup(F): FldFin -> GrpAb, Map`

Given $F = \mathbf{F}_q$, create the multiplicative group of R as an abelian group. This returns the (additive) cyclic group A of order $q-1$, together with a map from A to $F \setminus 0$, sending 1 to a primitive element of F.

`Set(F): FldFin -> SetEnum`

Create the enumerated set consisting of the elements of finite field F.

`VectorSpace(F, E): FldFin, FldFin -> ModTupFld, Map`

Given a finite field F that is an extension of degree n of E, define the natural isomorphism between F and the n-dimensional vector space E^n. The function returns two values:

(a) A vector space $V \cong E^n$;

(b) The isomorphism $\phi : F \to V$.

The basis of V is chosen to correspond with the power basis α^0, α^1, ..., α^{n-1} of F, where α is the generator returned by `Generator(F, E)`, so that $V = E \cdot 1 \times E \cdot \alpha \times \cdots \times E \cdot \alpha^{n-1}$ and $\phi : \alpha^i \to e_{i+1}$, (for $i = 0, \ldots, n-1$), where e_i is the basis vector of V having all components zero, except the i-th, which is one.

`VectorSpace(F, E, B): FldFin, FldFin, [ FldFinElt ] -> ModTupFld, Map`

Given a finite field F that is an extension of degree n of E, define the isomorphism between F and the n-dimensional vector space E^n defined by the basis B for F over E. The function returns two values:

(a) A vector space $V \cong E^n$;

(b) The isomorphism $\phi : F \to V$.

The basis of V is chosen to correspond with the basis $B = \beta_1, \beta_2, \ldots, \beta_n$ of F over E, as specified by the user, so that $V = E \cdot \beta_1 \times E \cdot \beta_2 \times \cdots \times E \cdot \beta_n$. $\phi : \beta_i \to e_i$, (for $i = 1, \ldots, n$), where e_i is the basis vector of V having all components zero, except the i-th, which is one.

`MatrixAlgebra(F, E): FldFin, FldFin -> AlgMat, Map`

Let F be a finite field that is an extension of degree n of E. The function returns two values:

(a) A matrix algebra A of degree n, such that A is isomorphic to F;

(b) An isomorphism $\phi : F \to A$.

The matrix algebra A will be the subalgebra of the full algebra of $n \times n$ matrices over E generated by the companion matrix C of the defining polynomial of F over E. The generator `Generator(F, E)` of F over E is thus mapped to C.

`MatrixAlgebra(A, E): AlgMat, FldFin -> AlgMat, Map`

Let F be a finite field. Let A be a matrix algebra over F, and E be a subfield of F. The function returns two values:

(a) A matrix algebra N over E isomorphic to A, obtained from A by expanding each component of an element of A into the block matrix associated with it;

(b) An E-isomorphism $\phi : A \to N$.

The matrix algebra N is A considered as an E-matrix algebra.

Example: Finite Field as a Vector Space `(ex-c6682e)`

Given the field F of 7^4 elements defined as an extension of the field $F49$ of 7^2 elements as above, we can construct two vector spaces, one of dimension 2, and the other of dimension 4:

```
> F7 := FiniteField(7);
> F49<w> := ext< F7 | 2 >;
> F<z> := ext< F49 | 2 >;
> v2, i2 := VectorSpace(F, F49);
> v2;
Full Vector space of degree 2 over GF(7^2)
> i2(z^12);
(   w w^28)
> v4, i4 := VectorSpace(F, PrimeField(F));
> v4;
Full Vector space of degree 4 over GF(7)
> i4(z^12);
(5 3 6 4)
```

`GaloisGroup(K, k): FldFin, FldFin -> GrpPerm, [FldFinElt]`

Compute the Galois group (which is of course cyclic) of K/k as a permutation group. The group is returned as well as the roots of the defining polynomial of K/k in a compatible ordering.

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

Computes the (cyclic) group of k-automorphisms of K. The group is returned as well as a sequence of all automorphisms and a map sending an element of the abstract automorphism group to an explicit automorphism.

22.3.2 Numerical Invariants

`Characteristic(F): FldFin -> RngIntElt`

`# F: FldFin -> FldFinElt`

`Degree(F): FldFin -> RngIntElt`

The absolute degree of F, that is, the degree over its prime subfield.

`Degree(F, E): FldFin, FldFin -> RngIntElt`

Given a finite field F that has been constructed as an extension of a field E, return the degree of F over E.

22.3.3 Defining Polynomial

`DefiningPolynomial(F): FldFin -> RngUPolElt`

Given a finite field F that has been constructed as an extension of a field E, return the polynomial with coefficients in E that was used to define F as an extension of E. This is the minimum polynomial of `F.1`.

`DefiningPolynomial(F, E): FldFin -> RngUPolElt`

Given a finite field F and a subfield E, return the polynomial with coefficients in E used to define F as an extension of E. This is the same as the minimum polynomial of the generator `Generator(F, E)` over E.

22.3.4 Ring Predicates and Booleans

`IsConway(F): FldFin -> BoolElt`

Given a finite field F, this function returns `true` iff F is defined over its prime field using a Conway polynomial.

`IsDefault(F): FldFin -> BoolElt`

Given a finite field F, this function returns `true` iff F is a default field.

`IsCommutative(F): FldFin -> BoolElt`

`IsUnitary(F): FldFin -> BoolElt`

`IsFinite(F): FldFin -> BoolElt`

`IsOrdered(F): FldFin -> BoolElt`

`IsField(F): FldFin -> BoolElt`

`IsEuclideanDomain(F): FldFin -> BoolElt`

`IsPID(F): FldFin -> BoolElt`

`IsUFD(F): FldFin -> BoolElt`

`IsDivisionRing(F): FldFin -> BoolElt`

`IsEuclideanRing(F): FldFin -> BoolElt`

`IsPrincipalIdealRing(F): FldFin -> BoolElt`

`IsDomain(F): FldFin -> BoolElt`

`F eq G: FldFin, Rng -> BoolElt`

`F ne G: FldFin, Rng -> BoolElt`

22.3.5 Roots

`Roots(f): RngUPolElt -> [ < FldFinElt, RngIntElt> ]`

Given a polynomial f over a finite field F, this function finds all roots of f in F, and returns a sorted sequence of tuples (pairs), each consisting of a root of f in F and its multiplicity.

`RootsInSplittingField(f): RngUPolElt[FldFin] -> [<RngUPolElt, RngIntElt>], FldFin`

Given a univariate polynomial f over a finite field K, compute the minimal splitting field S of f as an extension field of K, and return the roots of f in S, together with S. Using this function will be faster than computing the roots of f anew over the splitting field.

`FactorizationOverSplittingField(f): RngUPolElt[FldFin] -> [<RngUPolElt, RngIntElt>], FldFin`

`FactorisationOverSplittingField(f): RngUPolElt[FldFin] -> [<RngUPolElt, RngIntElt>], FldFin`

Given a univariate polynomial f over a finite field K, compute the minimal splitting field S of f as an extension field of K, and return the factorization (into linears) of f over S, together with S. Using this function will be faster than factorizing f anew over the splitting field.

`RootOfUnity(n, K): RngIntElt, FldFin -> FldFinElt`

Return a primitive n-th root of unity in the smallest possible extension field of K.

Example: Roots in a Splitting Field (ex-f06af7)

We compute the roots of a certain degree-20 polynomial f in its minimal splitting field.

```
> K := GF(2);
> P<x> := PolynomialRing(GF(2));
> f := x^20 + x^11 + 1;
> Factorization(f);
[
    <x^3 + x^2 + 1, 1>,
    <x^8 + x^7 + x^3 + x^2 + 1, 1>,
    <x^9 + x^7 + x^6 + x^4 + 1, 1>
]
> time r, S<w> := RootsInSplittingField(f);
Time: 0.040
```

We note that the splitting field S has degree 72 and there are 20 roots of f in S of course. We check that the evaluation of f at each root is zero.

```
> S;
Finite field of size 2^72
> DefiningPolynomial(S);
x^72 + x^48 + x^47 + x^44 + x^38 + x^35 + x^32 + x^31 + x^30 +
    x^29 + x^27 + x^25 + x^23 + x^22 + x^21 + x^18 + x^15 +
    x^12 + x^8 + x^4 + 1
> #r;
20
> r[1];
<w^68 + w^67 + w^64 + w^62 + w^60 + w^59 + w^56 + w^50 + w^49 +
    w^48 + w^47 + w^44 + w^43 + w^39 + w^37 + w^35 + w^33 + w^32
    + w^30 + w^29 + w^28 + w^25 + w^21 + w^19 + w^18 + w^16 +
    w^15 + w^14 + w^12 + w^10 + w^6 + w, 1>
> [IsZero(Evaluate(f, t[1])): t in r];
[ true, true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true ]
```

22.4 Element Operations

See also Section *Generic Element Functions*.

22.4.1 Arithmetic Operators

`+ a: FldFinElt -> FldFinElt`

`- a: FldFinElt -> FldFinElt`

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

`a - b: FldFinElt, FldFinElt -> FldFinElt`

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

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

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

```
a +:= b: FldFinElt, FldFinElt -> FldFinElt
a -:= b: FldFinElt, FldFinElt -> FldFinElt
a *:= b: FldFinElt, FldFinElt -> FldFinElt
```

22.4.2 Equality and Membership

```
a eq b: FldFinElt, FldFinElt -> BoolElt
a ne b: FldFinElt, FldFinElt -> BoolElt
a in F: FldFinElt, Rng -> BoolElt
a notin F: FldFinElt, Rng -> BoolElt
```

22.4.3 Parent and Category

```
Parent(a): FldFinElt -> FldFin
Category(a): FldFinElt -> Cat
```

22.4.4 Predicates on Ring Elements

```
IsZero(a): FldFinElt -> BoolElt
IsOne(a): FldFinElt -> BoolElt
IsMinusOne(a): FldFinElt -> BoolElt
IsNilpotent(a): FldFinElt -> BoolElt
IsIdempotent(a): FldFinElt -> BoolElt
IsUnit(a): FldFinElt -> BoolElt
IsZeroDivisor(a): FldFinElt -> BoolElt
IsRegular(a): FldFin -> BoolElt
IsIrreducible(a): FldFinElt -> BoolElt
IsPrime(a): FldFinElt -> BoolElt
IsPrimitive(a): FldFinElt -> BoolElt
```

Returns `true` if and only if the element a of F is a primitive element for F (i.e., if and only if the multiplicative order of a is $\#F - 1$).

```
IsPrimitive(f): RngUPolElt -> BoolElt
```

Given a univariate polynomial $f \in F[x]$, over a finite field F, such that the degree of f is greater than or equal to 1, this function returns `true` if and only if f defines a primitive extension $G = F[x]/f$ of F (that is, x is primitive in G).

```
IsNormal(a): FldFinElt -> BoolElt
```

Returns `true` if and only if the element a of F generates a normal basis for the field over the ground field, that is, if and only if $a, a^q, \ldots, a^{q^{n-1}}$ form a basis for F over the ground field $G = \mathbf{F}_q$.

```
IsNormal(a, E): FldFinElt -> BoolElt
```

Returns `true` if and only if the element a of the finite field F with q^n elements generates a normal basis for F over its subfield E, that is, if and only if $a, a^q, \ldots, a^{q^{n-1}}$ form a basis for F over E for $q = \#E$.

```
IsSquare(a): FldFinElt -> BoolElt
```

Given a finite field element $a \in F$, this function returns either `true` and an element $b \in F$ such that $b^2 = a$, or it returns `false` in the case that such an element does not exist.

22.4.5 Minimal and Characteristic Polynomial

`MinimalPolynomial(a): FldFinElt -> RngUPolElt`

The minimal polynomial of the element a of the field F, relative to the ground field of F. This is the unique minimal-degree monic polynomial with coefficients in the ground field, having a as a root.

`MinimalPolynomial(a, E): FldFinElt, FldFin -> RngUPolElt`

The minimal polynomial of the element a of the field F, relative to the subfield E of F. This is the unique minimal-degree monic polynomial with coefficients in E, having a as a root.

`CharacteristicPolynomial(a): FldFinElt -> RngUPolElt`

Given an element a of a finite field F, return the characteristic polynomial of a with respect to the ground field of F. (This polynomial is the characteristic polynomial of the companion matrix of a written as a polynomial over the ground field, and is a power of the minimal polynomial.)

`CharacteristicPolynomial(a, E): FldFinElt, FldFin -> RngUPolElt`

Given an element a of a finite field F, return the characteristic polynomial of a with respect to the subfield E of F. (This polynomial is the characteristic polynomial of the companion matrix of a written as a polynomial over E, and is a power of the minimal polynomial over E.)

22.4.6 Norm, Trace, and Frobenius

`Norm(a): FldFinElt -> FldFinElt`

The norm of the element a from the field F to the ground field of F.

`Norm(a, E): FldFinElt, FldFin -> FldFinElt`

The relative norm of the element a from the field F, with respect to the subfield E of F. The result is an element of E.

`AbsoluteNorm(a): FldFinElt -> FldFinElt`

`NormAbs(a): FldFinElt -> FldFinElt`

The absolute norm of the element a, that is, the norm to the prime subfield of the parent field F of a.

`Trace(a): FldFinElt -> FldFinElt`

The trace of the element a from the field F to the ground field of F.

`Trace(a, E): FldFinElt, FldFin -> FldFinElt`

The relative trace of the element a from field F, with respect to the subfield E of F. The result is an element of E.

`AbsoluteTrace(a): FldFinElt -> FldFinElt`

`TraceAbs(a): FldFinElt -> FldFinElt`

The trace of the element a, that is, the trace to the prime subfield of the parent field F of a.

`Frobenius(a): FldFinElt -> FldFinElt`

The Frobenius image of a w.r.t. the ground field of K; i.e., $a^{\#G}$, where G is the ground field of the parent of a.

`Frobenius(a, r): FldFinElt, RngIntElt -> FldFinElt`

The r-th Frobenius image of a w.r.t. the ground field of K; i.e., $a^{(\#G)^r}$, where G is the ground field of the parent of a.

`Frobenius(a, E): FldFinElt, FldFin -> FldFinElt`

The Frobenius image of x w.r.t. E; i.e., $x^{\#E}$.

`Frobenius(a, E, r): FldFinElt, FldFin, RngIntElt -> FldFinElt`

The Frobenius image of x w.r.t. E; i.e., $x^{(\#E)^r}$.

`NormEquation(K, y): FldFin, FldFin -> BoolElt, FldFinElt`

Given a finite field K and an element y of a subfield S of K, return whether an element $x \in K$ exists such that $\mathrm{Norm}(x, S) = y$, and, if so, such an element x (in K).

`Hilbert90(a, q): FldFinElt, RngIntElt -> FldFinElt`

Given an element a of some finite field k and a power q of the characteristic of k, return a solution of the Hilbert 90 equation $x^q x^{-1} = a$. Note that the solution may be in a finite-degree extension of k.

`AdditiveHilbert90(a, q): FldFinElt, RngIntElt -> FldFinElt`

Given an element a of some finite field k and a power q of the characteristic of k, return a solution of the additive Hilbert 90 equation $x^q - x = a$. Note that the solution may be in a finite-degree extension of k.

22.4.7 Order and Roots

`Order(a): FldFinElt -> RngIntElt`

The multiplicative order of the non-zero element a of the field F.

`FactoredOrder(a): FldFinElt -> RngIntElt`

The multiplicative order of the non-zero element a of the field F as a factorization sequence.

`SquareRoot(a): FldFinElt -> FldFinElt`

`Sqrt(a): FldFinElt -> FldFinElt`

The square root of the non-zero element a from the field F, i.e., an element y of F such that $y^2 = a$. An error results if a is not a square.

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

The n-th root of the non-zero element a from the field F, i.e., an element y of F such that $y^n = a$. An error results if no such root exists.

`IsPower(a, n): FldFinElt, RngIntElt -> BoolElt, FldFinElt`

Given a finite field element $a \in F$, and an integer $n > 0$, this function returns either `true` and an element $b \in F$ such that $b^n = a$, or it returns `false` in the case that such an element does not exist.

`AllRoots(a, n): FldFinElt, RngIntElt -> SeqEnum`

Given a finite field element $a \in F$, and an integer $n > 0$, return a sequence containing all of the n-th roots of a which lie in the same field F.

Example: Roots and Traces of Elements (ex-ead4e7)

Given the fields F and $F49$ defined above, we can use the following functions:

```
> F7 := FiniteField(7);
> F49<w> := ext< F7 | 2 >;
> F<z> := ext< F49 | 2 >;
> Root(z^73, 7);
z^1039
> Trace(z^73);
1
> Trace(z^73, F49);
w^44
> Norm(z^73);
3
> Norm(z^73, F49);
w^37
> Norm(w^37);
3
```

(continues on next page)

(continued from previous page)

```
> MinimalPolynomial(z^73);
x^2 + w^20*x + w^43
> MinimalPolynomial(z^73, F7);
x^4 + 4*x^2 + 4*x + 3
```

We now demonstrate the `NormEquation` function.

```
> Norm(z);
3
> NormEquation(F, F7!3);
true z
> Norm(z^30, F49);
w^30
> Parent(z) eq F;
true
> NormEquation(F, w^30);
true z^30
```

22.5 Polynomials for Finite Fields

`IrreduciblePolynomial(F, n): FldFin, RngIntElt -> RngUPolElt`

Given a finite field F and a positive integer $n > 1$, return a polynomial of degree n that is irreducible over F. If a Conway polynomial or a sparse polynomial is available, then it is returned.

`RandomIrreduciblePolynomial(F, n): FldFin, RngIntElt -> RngUPolElt`

Given a finite field F and a positive integer $n > 1$, return a random irreducible polynomial of degree n that is irreducible over F. The polynomial will be dense in general (that is, a Conway or stored sparse polynomial is not used).

`IrreducibleLowTermGF2Polynomial(n): RngIntElt -> RngUPolElt`

Given an integer n in the range $1 \leq n \leq 100000$, return the irreducible polynomial f of the form $x^n + g$ where the degree of g is minimal and g is the first such polynomial in lexicographical order. This uses a database of low-term irreducible polynomials over $\mathbf{F}_2$, constructed by Allan Steel in 2004 (thanks are expressed to William Stein for providing machines for some of the computations).

`IrreducibleSparseGF2Polynomial(n): RngIntElt -> RngUPolElt`

Given an integer n in the range $4 \leq n \leq 12800$, return the irreducible polynomial f of the form $x^n + g$ where g has 2 non-zero terms if possible and 4 non-zero terms if not; g is the first such polynomial in lexicographical order in either case. This uses a database of sparse irreducible polynomials over $\mathbf{F}_2$ constructed by Allan Steel in 1998.

`PrimitivePolynomial(F, m): FldFin, RngIntElt -> RngUPolElt`

Given a finite field F and a positive integer $m > 1$, construct a polynomial f of degree m that is primitive over F. Thus, f is irreducible over F, and it has a primitive root of the degree m extension field of F as a root.

`AllIrreduciblePolynomials(F, m): FldFin, RngIntElt -> { RngUPolElt }`

Given a finite field F and a positive integer $m > 1$, construct the set of all monic polynomials of degree m that are irreducible over F.

`ConwayPolynomial(p, n): RngIntElt, RngIntElt -> RngUPolElt`

Given a prime p and an exponent $n \geq 1$, return the Conway polynomial of degree n over $\mathbf{F}_p$. The Conway polynomial is defined in the introduction. Note that this polynomial is read in from a table containing Conway polynomials for a limited range of p, n only.

`ExistsConwayPolynomial(p, n): RngIntElt, RngIntElt -> BoolElt, RngUPolElt`

Given a prime p and an exponent $n \geq 1$, return `true` and the Conway polynomial for the field $\mathbf{F}_{p^n}$

if it is present in the table of Conway polynomials, which covers a limited range of p, n only (see `ConwayPolynomial` above), `false` otherwise.

22.6 Discrete Logarithms

Let K be a field of cardinality $q = p^k$, with p prime. Magma contains several advanced algorithms for computing discrete logarithms of elements of K. The two main kinds of algorithms used are as follows: (1) *Pohlig-Hellman* [PH78]: The running time is usually proportional to the square root of the largest prime l dividing $q - 1$; this is combined with the Shanks baby-step/giant-step algorithm (when l is very small) or the Pollard-ρ algorithm. (2) *Index-Calculus*: There is first a precomputation stage which computes and stores all the logarithms of a *factor base* (all the elements of the field corresponding to irreducible polynomials up to some bound) and then each subsequent individual logarithm is computed by expressing the given element in terms of the factor base.

The different kinds of finite fields in Magma are handled as follows (in this order):

(a) *Small Fields (any characteristic)*:

- If the largest prime l dividing $q - 1$ is reasonably small (typically, less than 2^{36}), the Pohlig-Hellman algorithm is used (the characteristic p is irrelevant).

(b) *Large Prime* :

- Suppose K is a prime field (so $q = p$). Then the *Gaussian integer sieve* [COS86, LO91a] is used if p has at least 4 bits but no more than 400 bits, $p-1$ is not a square, and one of the following is a quadratic residue modulo p: -1, -2, -3, -7, or -11. If the Gaussian integer sieve cannot be used and if p is no more than 300-bits, then the *linear sieve* [COS86, LO91a] is used. The precomputation stage always takes place and typically requires a lot more time than for computing individual logarithms (and may also require a lot of memory for large fields). Thus, the first call to the function `Log` below may take much more time than for subsequent calls. Also, for large prime fields, in comparison to the Gaussian method the linear sieve requires much more time and memory than the Gaussian method for the precomputation stage, and therefore it is only used when the Gaussian integer algorithm cannot be used. See the example `Discrete Logarithms by Linear Sieve (ex-a75220)` in the chapter on sparse matrices for an explanation of the basic linear sieve algorithm and for more information on the sparse linear algebra techniques employed.

(c) *Small Characteristic, Non-prime* :

- Since V2.19, if K is a finite field of characteristic p, where either $p \leq 7$ or a table as described below is present for K (such tables exist only for p less than about 100), then an implementation by Allan Steel of Coppersmith's index-calculus algorithm [Cop84, GM93, Tho01] is used. (Strictly speaking, Coppersmith's algorithm is for the case $p = 2$ only, but a straightforward generalization is used when $p > 2$.) A suite of external auxiliary tables boost the algorithm so that the precomputation stage computation to determine the logarithms of a factor base can be avoided for a large number of fields of small characteristic. This means that logarithms of individual elements can be computed immediately if a relevant table is present for the specific field; for $p > 7$ the algorithm is only used when such a table is present. By default, tables are included in the standard Magma distribution at least for all fields of characteristic 2, 3, 5, or 7 with cardinality up to 2^{200}. The user can optionally download a much larger suite of tables from the Magma optional downloads page `http://magma.maths.usyd.edu.au/magma/download/db/` (files `FldFinLog_2.tar.gz`, etc.; about 5GB total).

(d) *Medium Characteristic, Non-prime* :

- Since V2.30, if K is a non-prime field of characteristic p, where $7 < p < 2^{20}$, with q at least 2^{32} elements, the largest prime dividing $q - 1$ is at least 2^{37} (so that the Pohlig-Hellman algorithm is impractical), and no table as in (c) is available for K, then, apart from a few special cases, an implementation by Allan Steel of the *function field sieve* (FFS) [AH99, JL06] is used. This covers in particular all such fields with $p > 100$, which previously could only be handled by the Pohlig-Hellman algorithm. Two polynomials g_1 and g_2 of small degrees d_1 and d_2 over $\mathbf{F}_p$ are chosen such that $g_2(g_1(x)) - x$ and $g_1(g_2(x)) - x$ both have a factor of degree k, giving two representations of K between which there is an explicit isomorphism; the factor base consists of the monic irreducible polynomials over $\mathbf{F}_p$ of degree at most a bound B on each side, and relations are found by a lattice

sieve. The parameters B, d_1, d_2, and a limit on the size of the factor base are chosen automatically: by a rule calibrated on extensive timings, together with a table of measured optimal parameters for about 140 fields (with $101 \leq p \leq 211$ and $13 \leq k \leq 25$). Typically $B = 1$ is chosen for k up to about 19 and $B = 2$ beyond that. As in (b), the precomputation stage takes place once for the given p and k (and is then used for any field of cardinality p^k, whatever its defining polynomial), and takes much longer than the computation of subsequent individual logarithms. For example, the precomputation for $\mathbf{F}_{101^{13}}$ or $\mathbf{F}_{1013^9}$ takes a few seconds, and that for $\mathbf{F}_{223^{21}}$ takes several minutes.

(e) *Large Characteristic, Non-prime* :

- In all other cases, the Pohlig-Hellman algorithm is used.

`Log(x): FldFinElt -> RngIntElt`

The discrete logarithm of the non-zero element x from the field F, i.e., the unique integer k such that $x = w^k$ and $0 \leq k < (\#F - 1)$, where w is the primitive element of F (as returned by `PrimitiveElement`). Default parameters are automatically chosen if an index-calculus method is used (use *Sieve* below to set parameters). See also the procedure *SetPrimitiveElement*.

`Log(b, x): FldFinElt, FldFinElt -> RngIntElt`

The discrete logarithm to the base b of the non-zero element x from the field F, i.e., the unique integer k such that $x = b^k$ and $0 \leq k < (\#F - 1)$. If b is not a primitive element, then in some unusual cases the algorithm may take much longer than normal.

`ZechLog(K, n): FldFin, RngIntElt -> RngIntElt`

The Zech logarithm $Z(n)$ of the integer n for the field F, which equals the logarithm to base w of $w^n + 1$, where w is the primitive element of F. If w^n is the minus one element of K, then -1 is returned.

`Sieve(K): FldFin`

`Lanczos: BoolElt`	`Default: false`

(Procedure.) Call the Gaussian integer sieve on the prime finite field K if possible; otherwise call the linear sieve on K (assuming K is not too small).

If the parameter `Lanczos` is set to `true`, then the *Lanczos* algorithm [LO91b, Sec. 3] will be used for the linear algebra phase. This is generally *very much slower* than the default method (often 10 to 50 times slower), but it will take considerably less memory, so may be preferable for extremely large fields. See also the function `ModularSolution(A, M): MtrxSprs, RngIntElt → ModTupRng` in the chapter on sparse matrices for more information.

`SetVerbose("FFLog", v): MonStgElt, RngIntElt`

(Procedure.) Set the verbose printing level for the finite field logarithm algorithm to be v. Currently the legal values for v are 0, 1, and 2. If the level is 1, information is printed whenever the logarithm of an element is computed (unless the field is very small, in which case a lookup table is used). The value of 2 will print a very large amount of information.

Example: Discrete Logarithms in a Finite Field (ex-e6dfa5)

We demonstrate the `Log` function.

```
> F<z> := FiniteField(7^4);
> PrimitiveElement(F);
z;
> Log(z);
1
> Log(z^2);
2
> Log(z + 1);
419
```

(continues on next page)

(continued from previous page)

```
> z^419 eq z + 1;
true
> b := z + 1;
> b;
z^419
> Log(b, b);
1
> Log(b, z);
779
> b^779 eq z;
true
```

We now do similar things for a larger field of characteristic 2, which will use Coppersmith's algorithm to compute the logarithms.

```
> F<z> := GF(2, 73);
> Factorization(#F-1);
[ <439, 1>, <2298041, 1>, <9361973132609, 1> ]
> PrimitiveElement(F);
z
> time Log(z + 1);
429570031703221890 8392
Time: 5.400
> z^429570031703221890 8392;
z + 1
> time Log(z + 1);
429570031703221890 8392
Time: 0.000
> time Log(z^2);
2
Time: 0.000
> time Log(z^2134914112412412);
2134914112412412
Time: 0.000
> b := z + 1;
> b;
z + 1
> time Log(b, b);
1
Time: 0.010
> time Log(b, z);
2260630912967574270198
Time: 0.000
> b^2260630912967574270198;
z
```

22.7 Permutation Polynomials

Let K be a finite field. A polynomial representing (by the evaluation map) a bijection of K into itself is known as a *permutation polynomial*. The Dickson polynomials of the first and second kind are permutation polynomials when certain conditions are satisfied.

`DicksonFirst(n, a): RngIntElt, RngElt -> RngUPolElt`

Given a positive integer n, this function constructs the Dickson polynomial of the first kind $D_n(x, a)$

of degree n, where $D_n(x, a)$ is defined by

$$D_n(x,a) = \sum_{i=0}^{\lfloor n/2 \rfloor} \frac{n}{n-i} \binom{n-i}{i} (-a)^i x^{n-2i}.$$

`DicksonSecond(n, a): RngIntElt, RngElt -> RngUPolElt`

Given a positive integer n, this function constructs the Dickson polynomial of the second kind $E_n(x, a)$ of degree n, where $E_n(x, a)$ is defined by

$$E_n(x,a) = \sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n-i}{i} (-a)^i x^{n-2i}.$$

`IsProbablyPermutationPolynomial(p): RngUPolElt -> BoolElt`

NumAttempts: RngIntElt	Default: 100

Let p denote a polynomial defined over a finite field K. A probabilistic test is applied to determine whether the mapping on K defined by p is a bijection. The function returns `true` if the test succeeds for each of n attempts, otherwise `false`. By default, n is taken to be 100; a different value for n can be specified by use of the parameter `NumAttempts`.

Example: Dickson Permutation Polynomials (ex-41c38e)

Let K be a finite field of cardinality q. By a theorem of Nöbauer, the Dickson polynomial of the first kind of degree n is a permutation polynomial for K if and only if $(n, q^2 - 1) = 1$. Consider $K = \mathbf{F}_{16}$.

```
> Factorization(16^2 - 1);
[ <3, 1>, <5, 1>, <17, 1> ]
```

Thus, $D_n(x, a)$ will be a permutation polynomial for K providing that n is coprime to 3, 5, and 17.

```
> K<w> := GF(16);
> R<x> := PolynomialRing(K);
> a := w^5;
> p1 := DicksonFirst(3, a);
> p1;
x^3 + w^5*x
> #{ Evaluate(p1, x) : x in K };
11
> IsProbablyPermutationPolynomial(p1);
false
```

So $D_3(x, a)$ is not a permutation polynomial. However, $D_4(x, a)$ is a permutation polynomial:

```
> p1 := DicksonFirst(4, a);
> p1;
x^7 + w^5*x^5 + x
> #{ Evaluate(p1, x) : x in K };
16
> IsProbablyPermutationPolynomial(p1);
true
```

22.8 Bibliography

- [AH99] L. M. Adleman and M.-D. A. Huang. Function field sieve method for discrete logarithms over finite fields. *Inform. and Comput.*, 151(1–2):5–16, 1999.
- [BCS97] Wieb Bosma, John Cannon, and Allan Steel. Lattices of Compatibly Embedded Finite Fields. *J. Symbolic Comp.*, 24(3):351–369, 1997.
- [Cop84] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Trans. Inform. Theory*, IT–30(4):587–594, July 1984.
- [COS86] D. Coppersmith, A. M. Odlyzko, and R. Schroeppel. Discrete logarithms in GF(p). *Algorithmica*, 1:1–15, 1986.
- [GM93] D. M. Gordon and K. S. McCurley. Massively parallel computation of discrete logarithms. In Ernest F. Brickell, editor, *Advances in Cryptology—CRYPTO 1992*, volume 740 of LNCS, 312–323. Springer-Verlag, 1993. Proc. 12th Annual International Cryptology Conference, Santa Barbara, Ca, USA, August 16–20, 1992.
- [JL06] A. Joux and R. Lercier. The function field sieve in the medium prime case. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of Lecture Notes in Comput. Sci., 254–270. Berlin, 2006. Springer.
- [LO91a] B. A. LaMacchia and A. M. Odlyzko. Computation of Discrete Logarithms in Prime Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of INCS, 616–618. Springer-Verlag, 1991.
- [LO91b] B. A. LaMacchia and A. M. Odlyzko. Solving Large Sparse Linear Systems over Finite Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of LNCS, 109–133. Springer-Verlag, 1991.
- [PH78] S. C. Pohlig and M. E. Hellman. An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance. *IEEE Trans. Inform. Theory*, 24:106–110, 1978.
- [Tho01] Emmanuel Thomé. Computation of discrete logarithms in $\mathbf{F}_{2^{607}}$. In Colin Boyd and Ed Dawson, editors, *Advances in Cryptology—AsiaCrypt 2001*, volume 2248 of LNCS, 107–124. Springer-Verlag, 2001. Proc. 7th International Conference on the Theory and Applications of Cryptology and Information Security, Dec. 9–13, 2001, Gold Coast, Queensland, Australia.

Chapter 23

Nearfields

23.1 Introduction

In 1905, in the course of proving the independence of the field postulates, L. E. Dickson [Dic05a] (p. 203) introduced the first example of a nearfield. His example is a set of 9 elements with operations of addition and multiplication which satisfy all the axioms of a field except for the commutative law of multiplication and the right distributive law. Later that year Dickson [Dic05b] published a more extensive collection of examples: an infinite series obtained by twisting the multiplication of a Galois field and seven "irregular" examples.

The terminology 'nearfield' seems to have been introduced by Zassenhaus in his 1935 paper [Zas35] where he showed that the only finite nearfields (endliche Fastkörper) are those due to Dickson.

The irregular nearfields are often referred to as Zassenhaus nearfields and the nearfields in the infinite series are called Dickson nearfields.

In the papers of Dickson and Zassenhaus the nearfields are left-distributive but for the purposes of the Magma implementation we consider only right-distributive nearfields.

Nearfields are important in group theory, geometry, and a combination of these two fields. On the one hand, the finite sharply doubly transitive permutation groups are in one-to-one correspondence with the finite nearfields and on the other hand, nearfields coordinatise a class of translation planes [Hal59, Lun69] and they are the starting point for the construction of the Hughes planes [Dem71, Hug57]. Furthermore, every sharply transitive collineation group of projective space over a finite field is a quotient of the group of units of a nearfield [EK63] (see also, [Dem68] (§1.4, n^o 17)).

23.2 Nearfield Properties

A (right-distributive) *nearfield* is a set N containing elements 0 and 1 and with binary operations $+$ and $\circ$ such that

- **NF1:** $(N, +)$ is an abelian group and 0 is its identity element. Let $N^\times$ denote the set of non-zero elements of N.
- **NF2:** $(N^\times, \circ)$ is a group and 1 is its identity element.
- **NF3:** $a \circ 0 = 0 \circ a = 0$ for all $a \in N$.
- **NF4:** $(a + b) \circ c = a \circ c + b \circ c$ for all $a, b, c \in N$.

A subset S of a nearfield N is a *sub-nearfield* if $(S, +)$ and $(S \setminus \{0\}, \circ)$ are groups. The sub-nearfield *generated* by a subset X is the intersection of all sub-nearfields containing X. The *prime field* $\mathcal{P}(N)$ of N is the sub-nearfield generated by 1.

The inverse of $x \in N^\times$ is written $x^{[-1]}$. But where no confusion is possible we write multiplication of nearfield elements x and y as xy rather than $x \circ y$ and we write the inverse of x as x^{-1}. (In the Magma code we use "*" as the symbol for multiplication.)

If N is a finite nearfield, the prime field of N is a Galois field $\mathbf{F}_p$ for some prime p and p is the *characteristic* of N.

A nearfield of characteristic p is a vector space over its prime field and therefore its cardinality is p^n for some n. Every field is a nearfield.

If N is a nearfield, the *centre* of N is the set

$$\mathcal{Z}(N) = \{ x \in N \mid xy = yx \text{ for all } y \in N \}$$

and the *kernel* of N is the subfield

$$\mathcal{K}(N) = \{ x \in N \mid x(y+z) = xy + xz \text{ for all } y, z \in N \}.$$

It is clear that $\mathcal{Z}(N) \subseteq \mathcal{K}(N)$ but equality need not hold because, in general, $\mathcal{Z}(N)$ need not be closed under addition. Furthermore, the prime field $\mathcal{P}(N)$ need not be contained in $\mathcal{Z}(N)$. However, for the Dickson nearfields $\mathcal{Z}(N) = \mathcal{K}(N)$.

If N is a nearfield, then $\mathcal{Z}(N) = \bigcap\{\mathcal{K}(N)^x \mid x \in N, x \neq 0\}$.

23.2.1 Sharply Doubly Transitive Groups

A group G acting on a set Ω is *sharply doubly transitive* if G is doubly transitive on Ω and only the identity element fixes two points.

If G is a finite sharply doubly transitive group on Ω then

1. The set M consisting of the identity element and the elements of G without fixed points is an elementary abelian normal subgroup of G of order p^n for some n and some prime p.
2. Addition and multiplication between elements of Ω can be defined so that Ω becomes a nearfield and so that the group G is isomorphic to the group of all affine transformations $v \mapsto va + b$ of Ω, where $a \in \Omega^\times$ and $b \in \Omega$.

There is a converse to this theorem, namely if N is a nearfield, the group of all transformations $v \mapsto va+b$ acts sharply doubly transitively on N.

Let F be the prime field of N, regard N as a vector space over F and define $\mu : N^\times \to \mathrm{GL}(N)$ by $v^{\mu(a)} = va$. Then for all $a \in N^\times$, $a \neq 1$, the linear transformation $\mu(a)$ is fixed-point-free. Furthermore, μ defines an isomorphism between the multiplicative group $N^\times$ and its image in $\mathrm{GL}(N)$.

Suppose that $G = H \ltimes M$ is a sharply doubly transitive group of degree p^n, as above. The centre of G is trivial and M is a minimal normal subgroup. Thus if Ω' is a minimal permutation representation we may suppose that it is primitive. Then M is transitive on Ω' and since M is abelian, it acts regularly on Ω'. Thus p^n is the minimal degree of a faithful permutation representation of G.

23.3 Constructing Nearfields

There are two types of finite nearfield: the *regular* nearfields of Dickson and the *irregular* nearfields of Zassenhaus. In order to accommodate both types Magma has a 'virtual type' `Nfd` and types `NfdDck` and `NfdZss` which inherit from `Nfd`.

23.3.1 Dickson Nearfields

In order to begin exploring `Nfd` types in Magma we need a way to create instances of nearfields and their elements. As already mentioned there is a large class of nearfields first described by L. E. Dickson [Dic05a, Dic05b] in 1905 and in this section we describe how to construct them in Magma.

The nearfields resulting from this construction will be called *Dickson* (or *regular*) nearfields.

If p is a prime and if the positive integers h and v satisfy

- *if r is a prime or 4 and if r divides v, then r divides $p^h - 1$*

then (p, h, v) is a *Dickson triple*.

If we write $q = p^h$, the condition above is equivalent to

- *All prime factors of* v *divide* $q - 1$ *and* $q \equiv 3 \bmod 4$ *implies* $v \not\equiv 0 \bmod 4$.

We call (q, v) a *Dickson pair*.

`DicksonPairs(p, hlo, hhi, vlo, vhi): RngIntElt, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> SeqEnum`

The list of Dickson pairs (q, v) for prime p, where *hlo* and *hhi* are the lower and upper bounds on h and where *vlo* and *vhi* are the lower and upper bounds on v.

`DicksonPairs(p, h1, v1): RngIntElt, RngIntElt, RngIntElt -> SeqEnum`

The list of Dickson pairs (p^h, v) for the prime p, where $h1$ and $v1$ are upper bounds on h and v.

`DicksonTriples(p, hb, vb): RngIntElt, RngIntElt, RngIntElt -> SeqEnum`

The list of Dickson triples (p, h, v) for the prime p, where hb and vb are bounds on h and v.

Example: Dickson Pairs (ex-d7494c)

For each Dickson pair (equivalently Dickson triple), there is at least one Dickson nearfield.

```
> DicksonPairs(5,3,4,4,5);
[
    [ 125, 4 ],
    [ 625, 4 ]
]
> DicksonPairs(5,4,5);
[
    [ 5, 1 ],
    [ 5, 2 ],
    [ 5, 4 ],
    [ 25, 1 ],
    [ 25, 2 ],
    [ 25, 3 ],
    [ 25, 4 ],
    [ 125, 1 ],
    [ 125, 2 ],
    [ 125, 4 ],
    [ 625, 1 ],
    [ 625, 2 ],
    [ 625, 3 ],
    [ 625, 4 ]
]
> DicksonTriples(5,4,5);
[
    [ 5, 1, 1 ],
    [ 5, 1, 2 ],
    [ 5, 1, 4 ],
    [ 5, 2, 1 ],
    [ 5, 2, 2 ],
    [ 5, 2, 3 ],
    [ 5, 2, 4 ],
    [ 5, 3, 1 ],
    [ 5, 3, 2 ],
    [ 5, 3, 4 ],
    [ 5, 4, 1 ],
    [ 5, 4, 2 ],
    [ 5, 4, 3 ],
```

(continues on next page)

(continued from previous page)

```
    [ 5, 4, 4 ]
]
```

The isomorphism type of a Dickson nearfield depends on the choice of primitive element of the underlying Galois field. It has been shown by Lüneburg [Lun71] that if ϕ is the Euler phi-function and g is the order of p modulo v, there are $\phi(v)/g$ isomorphism classes of Dickson nearfields with the same Dickson triple (p, h, v).

The default nearfield will use the 'standard' primitive element of the field. The other variants with the same Dickson pair can be obtained by providing an integer s coprime to v. Internally this is converted to a suitable integer e coprime to $q^v - 1$ such that $s \equiv e \pmod{v}$.

NumberOfVariants(q, v): RngIntElt, RngIntElt -> RngIntElt

The number of non-isomorphic nearfields with Dickson pair (q, v).

NumberOfVariants(N): NfdDck -> RngIntElt

The number of variants of the Dickson nearfield N.

VariantRepresentatives(q, v): RngIntElt, RngIntElt -> SeqEnum

Representatives for the variant parameter of nearfields with Dickson pair (q, v).

Example: Variants of a Dickson Pair (ex-20e6ea)

For each Dickson pair there can be several variants. The variant representative can be used when constructing the corresponding Dickson nearfield.

```
> NumberOfVariants(625,4);
2
> VariantRepresentatives(625,4);
[ 1, 3 ]
```

DicksonNearfield(q, v : parameters): RngIntElt, RngIntElt -> NfdDck

```
Variant      : RngIntElt                    Default: 1
LargeMatrices: BoolElt                      Default: false
```

Create a Dickson nearfield from the Dickson pair (q, v). The **Variant** parameter is an integer s which can be used to specify the choice of primitive element (see the discussion following the intrinsic **DicksonTriples**). The parameter **LargeMatrices** is used only when the group of units of the nearfield is requested. The default is to represent the group of units as a matrix group defined over the kernel of the nearfield. But if **LargeMatrices** is **true**, the matrices are defined over the prime field.

Example: Constructing Dickson Nearfields (ex-da0033)

As indicated in the previous example, up to isomorphism, there are two Dickson nearfields with Dickson pair $(625, 4)$.

```
> D := DicksonNearfield(625,4);
> D3 := DicksonNearfield(625,4 : Variant := 3);
> D5 := DicksonNearfield(625,4 : Variant := 5);
> D eq D3;
false
> D3 eq D5;
false
> D eq D5;
true
> D;
Nearfield D of Dickson type defined by the pair (625, 4)
Order = 152587890625
```

23.3.2 Zassenhaus Nearfields

It was shown by Zassenhaus [Zas35] that in addition to the regular nearfields there are seven *irregular* nearfields. Zassenhaus gave constructions but did not prove their uniqueness. The proofs in [Zas35] are known to contain gaps. Perhaps the most reliable account of the existence and uniqueness of the irregular nearfields is the PhD thesis of Dancs-Groves [Gro74].

The seven finite nearfields which are not Dickson nearfields are the *Zassenhaus* nearfields.

Zassenhaus nearfields can be distinguished from regular nearfields by the fact that the multiplicative group of a finite nearfield N is metacyclic if and only if N is regular.

As a consequence, a Zassenhaus nearfield cannot occur as a subfield of a Dickson nearfield.

ZassenhausNearfield(n): RngIntElt -> NfdZss

Creates the nth Zassenhaus nearfield.

Example: Zassenhaus Nearfields (ex-af8523)

The orders of the Zassenhaus nearfields are 5^2, 11^2, 7^2, 23^2, 11^2, 29^2, and 59^2.

```
> for n := 1 to 7 do ZassenhausNearfield(n); end for;
Irregular nearfield Z with Zassenhaus number 1
Order = 25
Irregular nearfield Z with Zassenhaus number 2
Order = 121
Irregular nearfield Z with Zassenhaus number 3
Order = 49
Irregular nearfield Z with Zassenhaus number 4
Order = 529
Irregular nearfield Z with Zassenhaus number 5
Order = 121
Irregular nearfield Z with Zassenhaus number 6
Order = 841
Irregular nearfield Z with Zassenhaus number 7
Order = 3481
```

23.4 Operations on Elements

23.4.1 Nearfield Arithmetic

The operations of addition, subtraction, and negation are inherited from the underlying Galois field.

The operation of multiplication distinguishes a nearfield from a field. In a nearfield, multiplication is not commutative and the left distributive law fails.

+ a: NfdElt -> NfdElt

- a: NfdElt -> NfdElt

a + b: NfdElt, NfdElt -> NfdElt

a - b: NfdElt, NfdElt -> NfdElt

a * b: NfdElt, NfdElt -> NfdElt

a / b: NfdElt, NfdElt -> NfdElt

a ^ k: NfdElt, RngIntElt -> NfdElt

a +:= b: NfdElt, NfdElt -> NfdElt

a -:= b: NfdElt, NfdElt -> NfdElt

a *:= b: NfdElt, NfdElt -> NfdElt

Inverse(a): NfdElt -> NfdElt

The inverse of a.

23.4.2 Equality and Membership

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

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

`a in N: NfdElt, Rng -> BoolElt`

`a notin N: NfdElt, Rng -> BoolElt`

23.4.3 Parent and Category

`Parent(a): NfdElt -> FldFin`
`Category(a): NfdElt -> Cat`
`N ! x: Nfd, FldFinElt -> NfdElt`
`Element(N, x): Nfd, FldFinElt -> NfdElt`

Create a nearfield element from a finite field element.

`ElementToSequence(x): NfdElt -> SeqEnum`

Create a sequence from an element x of a nearfield.

23.4.4 Predicates on Nearfield Elements

`IsZero(a): NfdElt -> BoolElt`

`IsUnit(a): NfdElt -> BoolElt`

`IsIdentity(a): NfdElt -> BoolElt`

Example: Simple Nearfield Arithmetic (ex-bb693c)

This example illustrates some of the basic operations available on nearfields and their elements. There is a strong connection with the arithmetic of the underlying Galois field of a nearfield `D`, which is available as the attribute `D`gf`.

```
> D := DicksonNearfield(3^2,2);
> K := D`gf;
> x := Element(D,K.1);
> x;
$.1
> Parent(x);
Nearfield D of Dickson type defined by the pair (9, 2)
Order = 81
> x^2;
$.1^10
> Identity(D);
1
> assert x ne Identity(D);
> assert x eq x;
> Zero(D);
0
> Parent(Zero(D));
Nearfield D of Dickson type defined by the pair (9, 2)
Order = 81
> assert not IsZero(D!1);
> assert not IsZero(x);
> assert IsZero(Zero(D));
> K<z> := GF(3,4);
> x := Element(D,z^61);
```

(continues on next page)

(continued from previous page)

```
> y := Element(D,z^54);
> assert x + y eq Element(D,z^61+z^54);
> assert x - y eq Element(D,z^61-z^54);
> x*y;
z^35
> x/y;
z^7
> x^y;
z^29
```

Example: Failure of Left Distributivity (ex-60b080)

A nearfield is right-distributive, but unlike a Galois field, multiplication is not commutative and the left-distributive law may fail.

```
> N := DicksonNearfield(3^2,4);
> F<a> := N`gf;
> x := Element(N,a^5215);
> y := Element(N,a^5140);
> z := Element(N,a^5819);
> x*y eq y*x;
false
> x*(y+z) eq x*y+x*z;
false
> (y+z)*x eq y*x+z*x;
true
```

23.5 Operations on Nearfields

`N eq M: NfdDck, NfdDck -> BoolElt`

`N ne M: NfdDck, NfdDck -> BoolElt`

`#N: Nfd -> RngIntElt`

`Cardinality(N): Nfd -> RngIntElt`

The cardinality of the nearfield N.

`Random(N): Nfd -> NfdElt`

A random element of the nearfield N.

`Identity(N): Nfd -> NfdElt`

The multiplicative identity of the nearfield N.

`Zero(N): Nfd -> NfdElt`

The additive identity of the nearfield N.

`PrimeField(N): Nfd -> FldFin`

The prime field of the nearfield N.

`Kernel(N): Nfd -> FldFin`

Return the kernel of the nearfield N as a finite field.

23.6 The Group of Units

If N is a nearfield and $F = \mathcal{K}(N)$ is its kernel, N is a vector space over F and for all $u \in N^\times$, the map $x \mapsto x \circ u$ is an F-linear transformation. This action of $N^\times$ on the non-zero elements of the vector space is transitive and fixed-point-free.

Similarly, we may regard N as a vector space over its prime field and again the elements of $N^\times$ act as linear transformations. In the following code the vector space E could be either a vector space over the kernel or a vector space of the prime field. The default setting is to use the kernel. But if the parameter `LargeMatrices` is set to `true` when a regular nearfield is first defined, the prime field will be used. For irregular nearfields the kernel coincides with the prime field.

Let (p, h, v) be the Dickson triple for N, let ζ be a primitive element of $K = \mathbf{F}_{q^v}$ and put $A = \langle \zeta^v \rangle$. Then A is a group of order $m = (q^v - 1)/v$ and the elements $s_i = \zeta^{(q^i-1)/(q-1)}$ $(1 \le i \le v)$ are coset representatives for A in $K^\times$. Let Φ denote the Frobenius automorphism $x \mapsto x^q$ of K and define $\rho : K^\times \to \mathrm{Gal}(K/\mathbf{F}_p)$ by $\rho(u) = \Phi^i$ if $u \in s_i A$; that is, letting automorphisms of K act on the right, we have $x^{\rho(u)} = x^{q^i}$. The map ρ is not a homomorphism. However, its image is the cyclic group of order v generated by $\Phi = \rho(\zeta)$ and the fixed field of $\mathrm{im}\rho$ is $\mathbf{F}_q$; thus $\mathrm{im}\rho$ may be identified with $\mathrm{Gal}(K/\mathbf{F}_q)$.

The underlying set of N is identified with K and multiplication in N is defined to be $w \circ u = w^{\rho(u)} u$.

The group U of units of the Dickson nearfield $D = D(p, h, v, \zeta)$ has generators a and b and relations $a^m = 1$, $b^v = a^t$, and $b^{-1}ab = a^q$, where $q = p^h$, $m = (q^v - 1)/v$, and $t = m/(q - 1)$. Furthermore, Ellers and Karzel [EK64] show that $\gcd(v, t) = \gcd(q - 1, t) \le 2$. Equality holds if and only if $v \equiv 2 \pmod 4$ and $q \equiv 3 \pmod 4$ and this in turn is equivalent to the Sylow 2-subgroup of U being a generalised quaternion group.

The centre of D is $\mathbf{F}_q$ and its group of units is generated by ζ^{vt}.

`UnitGroup(N): Nfd -> GrpMat, Map`

`UnitGroup(GrpPerm, N): Nfd -> GrpPerm`

`UnitGroup(GrpPC, N): NfdDck -> GrpPC`

`UnitGroup(GrpPC, N): NfdZss -> GrpPC`

The unit group of the nearfield N.

Example: Unit Group of a Subnearfield `(ex-110490)`

In this example we construct the group of units of a subnearfield.

```
> N := DicksonNearfield(3^3,13);
> zeta := N`prim;
> x := N!(zeta^((3^39-1) div (3^13-1)));
> S := sub< N | x >;
> U := UnitGroup(S);
> IsAbelian(U);
true
> Factorisation(#N);
[ <3, 39> ]
> Factorisation(#S);
[ <3, 13> ]
> Factorisation(#Kernel(N));
[ <3, 3> ]
> S;
Nearfield S of Dickson type defined by the pair (1594323, 1)
Order = 1594323
```

`Order(x): NfdElt -> RngIntElt`

The order of the unit x of a nearfield.

As a matrix group, the unit group U of a nearfield acts regularly on the non-zero vectors of the underlying vector space E and consequently the affine group $E{\cdot}U$ is sharply two-transitive. All sharply two-transitive groups occur in this way.

`AffineGroup(N): Nfd -> GrpMat`

`AffineGroup(GrpPerm, N): Nfd -> GrpPerm`

`AffineGroup(GrpPC, N): NfdDck -> GrpPC`

AffineGroup(GrpPC, N): NfdZss -> GrpPC

The sharply two-transitive affine group associated with a nearfield, returned as a matrix group.

If $\Gamma = \mathrm{Gal}(K/\mathbf{F}_p)$ and $S = \Gamma \ltimes K^\times$ is the semidirect product of Γ and $K^\times$, then $D^\times \to S : w \mapsto \rho(w)w$ is an embedding of the multiplicative group $D^\times$ of $D = D(p, h, v, \zeta)$ in S, where multiplication in S is defined by

$$(\gamma_1 a_1)(\gamma_2 a_2) = \gamma_1\gamma_2 a_1^{\gamma_2} a_2.$$

If U is the image of $D^\times$ in S, then $\Gamma \cap U = 1$, $\Gamma U = S$, and $K^\times \cap U = A = \langle \zeta^v \rangle$. In fact, from the definition of ρ, we have $UK^\times = \Gamma_0 \ltimes K^\times$, where $\Gamma_0 = \mathrm{Gal}(K/\mathbf{F}_q)$. This is the *extended unit group* of the Dickson nearfield D.

ExtendedUnitGroup(D): NfdDck -> GrpMat

The extended unit group of a Dickson nearfield.

23.7 Automorphisms

IsIsomorphic(N1, N2): NfdDck, NfdDck -> BoolElt, Map

Test whether the regular nearfields N_1 and N_2 are isomorphic. If they are, return an isomorphism.

AutomorphismGroup(N): NfdDck -> GrpPerm, Map

AutomorphismGroup(N): NfdZss -> GrpPerm

The automorphism group A of the regular nearfield N and a map giving the action of A on N.

23.8 Nearfield Planes

A nearfield N is said to be *planar* if the mapping $x \mapsto -xa + xb$ is a permutation of N whenever $a \neq b$. Every finite nearfield is planar.

Given a finite nearfield N, there is an *affine* plane $\mathcal{A}$ with point set $N \times N$ and lines given by the equations

$$y = xm + b$$
$$x = c$$

Let $\mathcal{P}$ be the corresponding projective plane, obtained from $\mathcal{A}$ by adjoining a line L_∞ called the *line at infinity*. We label the points of $\mathcal{P}$ with triples of elements of N as follows.

(1) For every point (x, y) of $\mathcal{A}$ there is a point $[1, x, y]$ of $\mathcal{P}$.

(2) For every m there is an "ideal" point $[0, 1, m]$ of $\mathcal{P}$ which lies on every line $y = xm + b$ $(b \in N)$ and on L_∞.

(3)There is a point $[0, 0, 1]$ of $\mathcal{P}$ which lies on every line $x = c$ and on L_∞.

The lines of $\mathcal{P}$ may also be labelled by triples of elements of N: the line $y = xm + b$ corresponds to the triple $[-b, -m, 1]$ and the line $x = c$ corresponds to $[-c, 1, 0]$. The line L_∞ is labelled $[1, 0, 0]$. A point $\pi = [w, x, y]$ is incident with a line $L = [a, b, c]$ if and only if $wa + xb + yc = 0$.

Every collineation of $\mathcal{A}$ extends to a collineation of $\mathcal{P}$.

ProjectivePlane(N : parameters): Nfd -> PlaneProj, PlanePtSet, PlaneLnSet

Check: BoolElt	Default: false

The finite projective plane coordinatised by the nearfield N. The points of the nearfield plane are represented as triples of Galois field elements.

Example: Projective Plane of a Nearfield (ex-f4d8e0)

```
> N := DicksonNearfield(3,2);
> pl := ProjectivePlane(N);
> A := AutomorphismGroup(pl);
> #A;
311040
> CompositionFactors(A);
    G
    |  Cyclic(2)
    *
    |  Alternating(5)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(3)
    *
    |  Cyclic(3)
    *
    |  Cyclic(3)
    *
    |  Cyclic(3)
    1
```

23.8.1 Hughes Planes

In 1957 Hughes [Hug57] discovered a class of finite projective planes constructed from the Dickson nearfields which have rank 2 over their kernel. Neither these planes nor their duals are translation planes and therefore they cannot be obtained by the coordinatisation method of the previous section. Hughes' methods required the kernel to be central but in 1960 the construction was generalised by Rosati [Ros60] to include the Zassenhaus nearfields (see also Dembowski [Dem68, §5.4] and [Dem71]). For simplicity of notation we shall use the term 'Hughes plane' to include both Hughes planes and generalised Hughes planes.

HughesPlane(N : parameters): Nfd -> PlaneProj, PlanePtSet, PlaneLnSet

```
Check: BoolElt                    Default: false
```

The Hughes plane based on the nearfield N.

Example: Hughes Planes (ex-c201b6)

We construct the Desarguesian projective plane PG(2,49) and then, using nearfields of order 49, we construct three non-Desarguesian projective planes. These four planes can be distinguished by the orders of their collineation groups.

```
> DP := FiniteProjectivePlane(49); // Desarguesian plane
> DP;
Projective Plane PG(2, 49)
> CD  := CollineationGroup(DP);
> FactoredOrder(CD);
[ <2, 10>, <3, 3>, <5, 2>, <7, 6>, <19, 1>, <43, 1> ]
```

(continues on next page)

(continued from previous page)

```
> N := DicksonNearfield(7,2);
> NP := ProjectivePlane(N);
> NP;
Projective Plane of order 49
> CN := CollineationGroup(NP);
> FactoredOrder(CN);
[ <2, 10>, <3, 2>, <7, 4> ]
> Z := ZassenhausNearfield(3);
> #Z;
49
> ZP := ProjectivePlane(Z);
> CZ := CollineationGroup(ZP);
> FactoredOrder(CZ);
[ <2, 9>, <3, 3>, <7, 4> ]
> HP := HughesPlane(N);
> HP;
Projective Plane of order 49
> CH := CollineationGroup(HP);
> FactoredOrder(CH);
[ <2, 6>, <3, 3>, <7, 3>, <19, 1> ]
> CompositionFactors(CH);
    G
    |  Cyclic(3)
    *
    |  A(2, 7)               = L(3, 7)
    *
    |  Cyclic(2)
    1
```

23.9 Bibliography

- [Dem68] Peter Dembowski. *Finite geometries*. Ergebnisse der Mathematik und ihrer Grenzgebiete, Band 44. Springer-Verlag, Berlin, 1968.
- [Dem71] Peter Dembowski. Generalized Hughes planes. *Canad. J. Math.*, 23:481–494, 1971.
- [Dic05a] Leonard Eugene Dickson. Definitions of a group and a field by independent postulates. *Trans. Amer. Math. Soc.*, 6(2):198–204, 1905.
- [Dic05b] Leonard Eugene Dickson. On finite algebras. *Nachr. Kgl. Ges. Wiss. Göttingen, Math.-phy. Klasse*, pages 358–393, 1905.
- [EK63] Erich Ellers and Helmut Karzel. Kennzeichnung elliptischer Gruppenräume. *Abh. Math. Sem. Univ. Hamburg*, 26:55–77, 1963.
- [EK64] Erich Ellers and Helmut Karzel. Endliche Inzidenzgruppen. *Abh. Math. Sem. Univ. Hamburg*, 27:250–264, 1964.
- [Gro74] Susan Dancs Groves. *Locally finite near-fields*. PhD thesis, Australian National University, 1974.
- [Hal59] Marshall Hall, Jr. *The theory of groups*. The Macmillan Co., New York, N.Y., 1959.
- [Hug57] D. R. Hughes. A class of non-Desarguesian projective planes. *Canad. J. Math.*, 9:378–388, 1957.
- [Lun69] Heinz Lüneburg. *Lectures on projective planes*. University of Illinois, Chicago, 1969.
- [Lun71] Heinz Lüneburg. Über die Anzahl der Dickson'schen Fastkörper gegebener Ordnung. In *Atti del Convegno di Geometria Combinatoria e sue Applicazioni (Univ. Perugia, Perugia, 1970)*, pages 319–322. Ist. Mat., Univ. Perugia, Perugia, 1971.

- [Ros60] Luigi Antonio Rosati. Su una generalizzazione dei piani di Hughes. *Atti Accad. Naz. Lincei Rend. Cl. Sci. Fis. Mat. Nat. (8)*, 29:303–308 (1961), 1960.
- [Zas35] Hans Zassenhaus. Über endliche Fastkörper. *Abh. Math. Sem. Univ. Hamburg*, 11:187–220, 1935.

Chapter 24

Univariate Polynomial Rings

24.1 Introduction

Univariate polynomial rings may be defined over any ring R. Let us denote the univariate polynomial ring in indeterminate x over the coefficient ring R by $P = R[x]$. There are two kinds of polynomials in Magma: *univariate* polynomials, represented as vectors of coefficients; and *multivariate polynomials* represented in distributive form (linear sums of coefficient-monomial pairs). In this chapter we discuss univariate polynomials.

24.1.1 Representation

The vector representation enables fast arithmetic on univariate polynomials, but it requires considerable amounts of memory for multivariate polynomials; therefore, only univariate polynomial rings using the vector representation can be created directly (but, if one insists, it is possible to create univariate polynomial rings over univariate polynomial rings, etc.). Multivariate polynomials can be stored efficiently in distributive form, but the arithmetic operations on polynomials of one variable stored in this way may be considerably slower.

24.2 Creation Functions

24.2.1 Creation of Structures

There are two different ways to create polynomial rings, corresponding to the different internal representations (vector versus distributive — see the introductory section): `PolynomialRing(R)` and `PolynomialRing(R, n)`. The latter should be used to create multivariate polynomials; the former should be used for univariate polynomials.

`PolynomialAlgebra(R): Rng -> RngUPol`

`PolynomialRing(R): Rng -> RngUPol`

Global: BoolElt	Default: true

Create a univariate polynomial ring over the ring R. The ring is regarded as an R-algebra via the usual identification of elements of R and the constant polynomials. The polynomials are stored in vector form, which allows fast arithmetic. It is not recommended to use this function recursively to build multivariate polynomial rings. The angle bracket notation can be used to assign names to the indeterminate, e.g.: `P<x> := PolynomialRing(R)`. By default, the unique *global* univariate polynomial ring over R will be returned; if the parameter `Global` is set to `false`, then a non-global univariate polynomial ring over R will be returned (to which a separate name for the indeterminate can be assigned).

Example: Global and Non-Global Polynomial Rings (ex-2d4846)

We demonstrate the difference between global and non-global rings. We first create the global univariate polynomial ring over $\mathbb{Q}$ twice.

```
> Q := RationalField();
> P<x> := PolynomialRing(Q);
> PP := PolynomialRing(Q);
> P;
Univariate Polynomial Ring in x over Rational Field
> PP;
Univariate Polynomial Ring in x over Rational Field
> PP.1;
x
```

PP is identical to P. We now create non-global univariate polynomial rings (which are also different to the global polynomial ring P). Note that elements of all the rings are mathematically equal by automatic coercion.

```
> Pa<a> := PolynomialRing(Q: Global := false);
> Pb<b> := PolynomialRing(Q: Global := false);
> Pa;
Univariate Polynomial Ring in a over Rational Field
> Pb;
Univariate Polynomial Ring in b over Rational Field
> a;
a
> b;
b
> P;
Univariate Polynomial Ring in x over Rational Field
> x;
x
> x eq a; // Automatic coercion
true
> x + a;
2*x
```

24.2.2 Print Options

The `AssignNames` and `Name` functions can be used to associate a name with the indeterminate of a polynomial ring after creation.

`AssignNames(~P, s): RngUPol, [ MonStgElt ]`

Procedure to change the name of the indeterminate of a polynomial ring P. The indeterminate will be given the name of the string in the sequence s. This procedure only changes the name used in printing the elements of P. It does *not* assign to identifiers corresponding to the strings the indeterminates in P; to do this, use an assignment statement, or use angle brackets when creating the field. Note that since this is a procedure that modifies P, it is necessary to have a reference `~P` to P in the call to this function.

`Name(P, i): RngUPol, RngIntElt -> RngUPolElt`

Given a polynomial ring P, return the i-th indeterminate of P (as an element of P).

24.2.3 Creation of Elements

The easiest way to create polynomials in a given ring is to use the angle bracket construction to attach names to the indeterminates, and to use these names to express polynomials (see the examples). Below we list other options.

`P . 1: RngUPol, RngInt -> RngPolElt`

Return the indeterminate for the polynomial ring P, as an element of P.

`elt< P | a`$_0$`, ..., a_d >: RngUPol, RngElt, ..., RngElt -> RngUPolElt`

Given a polynomial ring $P = R[x]$ and elements $a_0, \ldots, a_d$ coercible into the coefficient ring R, return the polynomial $a_0 + a_1 x_n + \cdots + a_d x_n^d$ as an element of P.

`P ! s: RngUPol, RngElt -> RngPolElt`

`P ! s: RngUPol, [ RngElt ] -> RngPolElt`

`elt< P | s >: RngUPol, [ RngElt ] -> RngUPolElt`

Coerce the element s into the polynomial ring $P = R[x]$. The following possibilities for s exist.

(a) s is an element of P: it is returned unchanged;

(b) s is an element of a ring that can be coerced into the coefficient ring R of P: the constant polynomial s is returned;

(c) $s = \sum_j s_j y^j$ is an element of a univariate polynomial ring whose coefficient ring elements s_j can be coerced into R: the polynomial $\sum_j r_j x^j$ is returned, where r_j is the result of coercing s_j into R;

(c) s is a sequence: if s is empty then the zero element of P is returned, and if it is non-empty but the elements of the sequence can be coerced into R then the polynomial $\sum_j s[j] x_n^{j-1}$ is returned.

Note that constant polynomials may be coerced into their coefficient rings.

`Polynomial(Q): [ RngElt ] -> RngUPolElt`

Given a sequence Q of elements from a ring R, create the polynomial over R whose coefficients are given by Q. This is equivalent to `PolynomialRing(Universe(Q))!Q`.

`Polynomial(R, Q): Rng, [ RngElt] -> RngUPolElt`

Given a ring R and sequence Q of elements from a ring S, create the polynomial over R whose coefficients are given by the elements of Q, coerced into S. This is equivalent to `PolynomialRing(R)!ChangeUniverse(Q, R)`.

`Polynomial(R, f): Rng, RngUPolElt -> RngUPolElt`

Given a ring R and a polynomial f over a ring S, create the polynomial over R obtained from f by coercing its coefficients into S. This is equivalent to `PolynomialRing(R)!f`.

`One(P): RngUPol -> RngUPolElt`

`Identity(P): RngUPol -> RngUPolElt`

`Zero(P): RngUPol -> RngUPolElt`

`Representative(P): RngUPol -> RngUPolElt`

Example: Creating Polynomials (ex-365cfe)

The easiest way to create the polynomial $x^3 + 3x + 1$ (over the integers) is as follows.

```
> P<x> := PolynomialRing(Integers());
> f := x^3+3*x+1;
> f;
x^3 + 3*x + 1
```

Alternative ways to create polynomials are given by the element constructor (rarely used) and the ! operator:

```
> P<x> := PolynomialAlgebra(Integers());
> f := elt< P | 2, 3, 0, 1 >;
> f;
x^3 + 3*x + 2
> P ! [ 2, 3, 0, 1 ];
x^3 + 3*x + 2
```

Note that it is important to realize that a sequence is coerced into a polynomial ring by coercing its entries into the coefficient ring, and it is not attempted first to coerce the sequence as a whole into the coefficient ring:

```
> Q := RationalField();
> Q ! [1, 2];
1/2
> P<x> := PolynomialRing(Q);
> P ! [1,2];
2*x + 1
> P ! Q ! [1,2];
1/2
> P ! [ [1,2], [2,3] ];
2/3*x + 1/2
```

24.3 Structure Operations

24.3.1 Related Structures

The main structure related to a polynomial ring is its coefficient ring. Univariate polynomial rings belong to the Magma category RngUPol.

BaseRing(P): RngUPol -> Rng

CoefficientRing(P): RngUPol -> Rng

CoefficientRing(f): RngUPolElt -> Rng

Return the coefficient ring of polynomial ring P (the parent of f).

Category(P): RngUPol -> Cat

Parent(P): RngUPol -> Pow

PrimeRing(P): RngUPol -> Rng

24.3.2 Changing Rings

The ChangeRing function enables changing coefficient rings on a polynomial ring.

ChangeRing(P, S): RngUPol, Rng -> RngUPol, Map

Given a polynomial ring $P = R[x]$, together with a ring S, construct the polynomial ring $Q = S[y]$, together with the homomorphism h from P to Q. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S. The homomorphism h will apply this coercion to the coefficients of elements in P to return elements of Q. The usual angle bracket notation can be used for indeterminate names on the result.

ChangeRing(P, S, f): RngUPol, Rng, Map -> RngUPol, Map

Given a polynomial ring $P = R[x]$, together with a ring S and a map $f\colon R \to S$, construct the polynomial ring $Q = S[y]$ together with the homomorphism h from P to Q obtained by applying h to the coefficients of elements of P. The usual angle bracket notation can be used for indeterminate names on the result.

Example: Changing the Coefficient Ring (ex-332ea1)

In the first example of `ChangeRing` below we use automatic coercion of integers to rationals to go from $\mathbb{Z}[x]$ to $\mathbb{Q}[y]$. In fact `!` can be used for this as well. In the second example we use a map to obtain a non-standard embedding (not mapping 1 to 1) of $\mathbb{Z}$ in $\mathbb{Q}$.

```
> Z := Integers();
> Q := RationalField();
> P<x> := PolynomialRing(Z);
> S<y>, h := ChangeRing(P, Q);
> h(x^3-2*x+5);
y^3 - 2*y + 5
> S ! (x^3-2*x+5);
y^3 - 2*y + 5
> m := hom< Z -> Q | x :-> 3*x >;
> S<y>, h := ChangeRing(P, Q, m);
> h(x^3-2*x+5);
3*y^3 - 6*y + 15
```

24.3.3 Numerical Invariants

The characteristic can be obtained for any polynomial ring, the rank for free polynomial rings, and the cardinality only for finite quotients.

`Rank(P): RngUPol -> RngIntElt`

Return the rank of the polynomial ring P, defined as the maximal number of independent indeterminates in P over its coefficient ring; for univariate polynomial rings this will therefore always return 1.

`# P: RngUPolRes -> RngIntElt`

Return the number of elements of P; this will only return an integer value if P is finite, which for polynomial rings can only happen for quotients of polynomial rings over finite coefficient rings.

`Characteristic(P): RngUPol -> RngIntElt`

24.3.4 Ring Predicates and Booleans

The usual ring functions returning Boolean values are available on polynomial rings.

`IsCommutative(P): RngUPol -> BoolElt`

`IsUnitary(P): RngUPol -> BoolElt`

`IsFinite(P): RngUPol -> BoolElt`

`IsOrdered(P): RngUPol -> BoolElt`

`IsField(P): RngUPol -> BoolElt`

`IsEuclideanDomain(P): RngUPol -> BoolElt`

`IsPID(P): RngUPol -> BoolElt`

`IsUFD(P): RngUPol -> BoolElt`

`IsDivisionRing(P): RngUPol -> BoolElt`

`IsEuclideanRing(P): RngUPol -> BoolElt`

`IsDomain(P): RngUPol -> BoolElt`

`IsPrincipalIdealRing(P): RngUPol -> BoolElt`

```
P eq Q: RngUPol, RngUPol -> BoolElt

P ne Q: RngUPol, RngUPol -> BoolElt

P lt Q: RngUPol, RngUPol -> BoolElt

P gt Q: RngUPol, RngUPol -> BoolElt

P le Q: RngUPol, RngUPol -> BoolElt

P ge Q: RngUPol, RngUPol -> BoolElt
```

24.3.5 Homomorphisms

A ring homomorphism taking a polynomial ring $R[x]$ as its domain requires 2 pieces of information, namely, a map (homomorphism) telling how to map the coefficient ring R, together with the image of the indeterminate x. The map may be omitted.

`hom< P -> S | f, y >: RngUPol, Rng, Map, RngElt -> Map`

`hom< P -> S | y >: RngPol, Rng, RngElt -> Map`

Given a polynomial ring $P = R[x]$, a ring S, a map $f : R \to S$, and an element $y \in S$, create the homomorphism $g : P \to S$ given by that $g(\sum s_i x^i) = \sum f(s_i)y^i$. 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 image y is allowed to be from a structure that allows automatic coercion into S.

Example: Homomorphism into the Real Field (ex-c83216)

In this example we map $\mathbb{Z}[x]$ into the reals by sending x to $1/2$. Note that we do not have a choice for the coefficient map (since we require it to be unitary), and also that we give the image of x as a rational number that is automatically coerced into the reals.

```
> Z := Integers();
> P<x> := PolynomialRing(Z);
> Re := RealField(20);
> half := hom< P -> Re | 1/2 >;
> half(x^3-3*x+5);
3.625
```

24.4 Element Operations

The categories for elements in univariate polynomial rings and their quotients are `RngUPolElt` and `RngUPolResElt`

24.4.1 Parent and Category

`Parent(p): RngUPolElt -> RngUPol`

`Category(p): RngUPolElt -> Cat`

24.4.2 Arithmetic Operators

The usual unary and binary ring operations are available for univariate polynomials, with the following notable restrictions. Since inverses cannot generally be obtained in polynomial rings, division (using `/`) of polynomials is not allowed, and neither are negative powers. For polynomial rings over fields division by elements of the coefficient field are allowed. The operators `div` and `mod` give results corresponding to the quotient and the remainder of division of the arguments. See the section on quotient and remainder for details.

```
+ a: RngUPolElt -> RngUPolElt

- a: RngUPolElt -> RngUPolElt

a + b: RngUPolElt, RngUPolElt -> RngUPolElt

a - b: RngUPolElt, RngUPolElt -> RngUPolElt

a * b: RngUPolElt, RngUPolElt -> RngUPolElt

a ^ k: RngUPolElt, RngIntElt -> RngUPolElt

a / b: RngUPolElt, RngElt -> FldFunUElt

a div b: RngUPolElt, RngUPolElt -> RngUPolElt

a mod b: RngUPolElt, RngUPolElt -> RngUPolElt

a +:= b: RngUPolElt, RngUPolElt -> RngUPolElt

a -:= b: RngUPolElt, RngUPolElt -> RngUPolElt

a *:= b: RngUPolElt, RngUPolElt -> RngUPolElt
```

24.4.3 Equality and Membership

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

a ne b: RngUPolElt, RngUPolElt -> BoolElt

a in R: RngUPolElt, Rng -> BoolElt

a notin R: RngUPolElt, Rng -> BoolElt
```

24.4.4 Predicates on Ring Elements

The list belows contains the general ring element predicates. Note that not all functions are available for every coefficient ring.

```
IsZero(a): RngUPolElt -> BoolElt

IsOne(a): RngUPolElt -> BoolElt

IsMinusOne(a): RngUPolElt -> BoolElt

IsNilpotent(a): RngUPolElt -> BoolElt

IsIdempotent(a): RngUPolElt -> BoolElt

IsUnit(a): RngUPolElt -> BoolElt

IsZeroDivisor(a): RngUPolElt -> BoolElt

IsRegular(a): RngUPolElt -> BoolElt

IsIrreducible(a): RngUPolElt -> BoolElt

IsPrime(a): RngUPolElt -> BoolElt

IsMonic(a): RngUPolElt -> BoolElt
```

24.4.5 Coefficients and Terms

`Coefficients(p): RngUPolElt -> [ RngElt ]`

`ElementToSequence(p): RngUPolElt -> [ RngElt ]`

`Eltseq(p): RngUPolElt -> [ RngElt ]`

The coefficients of the polynomial $p \in R[x]$ in ascending order, as a sequence of elements of R.

`Coefficient(p, i): RngUPolElt, RngIntElt -> RngElt`

Given a polynomial $p \in R[x]$ and an integer $i \geq 0$, return the coefficient of the i-th power of x in f. (If i exceeds the degree of f then zero is returned.) The return value is an element of R.

`MonomialCoefficient(p, m): RngUPolElt, RngUPolElt -> RngElt`

Given elements p and m of a polynomial ring $P = R[x]$, where m is a monomial (that is, has exactly one non-zero base coefficient, which must be 1), return the coefficient of m in p, as an element of the coefficient ring R.

`LeadingCoefficient(p): RngUPolElt -> RngElt`

Return the coefficient of the highest occurring power of x in $p \in R[x]$, as an element of the coefficient ring R.

`TrailingCoefficient(p): RngUPolElt -> RngElt`

Return the coefficient of the lowest occurring power of x in $p \in R[x]$, as an element of the coefficient ring R.

`ConstantCoefficient(p): RngUPolElt -> RngElt`

Return the constant term, i.e., the coefficient of x^0 as an element of the coefficient ring R.

`Terms(p): RngUPolElt -> [ RngUPolElt ]`

Return the non-zero terms of the polynomial $p \in P = R[x]$ in ascending order with respect to the degree, as a sequence of elements of P with ascending degrees.

`LeadingTerm(p): RngUPolElt -> RngUPolElt`

Return the term of $p \in P = R[x]$ with the highest occurring power of x, as an element of P. The coefficient of the result will be the leading coefficient of p.

`TrailingTerm(p): RngUPolElt -> RngUPolElt`

Return the term of $p \in P = R[x]$ with the lowest occurring power of x, as an element of P. The coefficient of the result will be the trailing coefficient of p.

`Monomials(p): RngUPolElt -> SeqEnum`

The monomials of the univariate p, matching up with `Coefficients(p)`, that is a sequence of powers of the indeterminate up to the degree of p.

`Support(p): RngUPolElt -> [RngIntElt], [RngElt]`

Given a polynomial $p \in R[x]$, return the positions in p for which there are non-zero coefficients, and the corresponding coefficients.

`Round(p): RngUPolElt -> RngUPolElt`

Given a polynomial $p \in P = R[x]$ where R is a subring of the real field (the ring of integers $\mathbb{Z}$, the rational field $\mathbb{Q}$, or a real field), return the polynomial in $Z[x]$ obtained from p by rounding all the coefficients of p.

`Valuation(p): RngUPolElt -> RngIntElt`

The valuation of a polynomial $p \in R[x]$, that is, the exponent of the largest power of x which divides p. Note that the zero polynomial has valuation ∞.

24.4.6 Degree

`Degree(p): RngUPolElt -> RngIntElt`

The degree of a polynomial $p \in R[x]$, that is, the exponent of the largest power of x that occurs with non-zero coefficient. Note that the zero polynomial has degree -1.

24.4.7 Roots

`Roots(p): RngUPolElt -> [ < RngElt, RngIntElt> ]`

```
Max: RngIntElt                    Default:
```

Given a polynomial p over one of a certain collection of coefficient rings, this function returns a sorted sequence of pairs of coefficient ring element and integer, where the ring element is a root of p in the coefficient ring, and the integer its multiplicity. Currently the coefficient rings that are allowed comprise complex and real fields, integers and rationals, finite fields and residue class rings with prime modulus. If the parameter `Max` is set to a non-negative number m, at most m roots are returned.

`Roots(p, S): RngUPolElt -> [ < RngElt, RngIntElt> ]`

Given a polynomial p over one of a certain collection of coefficient rings as well as a ring S into which the coefficients of p can be coerced automatically, this function returns a sorted sequence of pairs of ring element and integer, where the ring element is a root of p in the ring S, and the integer its multiplicity. Currently the coefficient rings that are allowed comprise complex and real fields, integers and rationals, finite fields and residue class rings with prime modulus.

`HasRoot(p): RngUPolElt -> BoolElt, RngElt`

Given a polynomial p over the coefficient ring R this function returns `true` iff p has a root in R. If the result is `true`, the function also returns a root of p as a second return value. Currently the coefficient rings that are allowed comprise complex and real fields, integers and rationals, finite fields and residue class rings with prime modulus. Note that particularly for finite fields, this method may be much faster than the computation of all roots of the polynomial.

`HasRoot(p, S): RngUPolElt, Rng -> BoolElt, RngElt`

Given a polynomial p over the coefficient ring R and a ring S which contains R, this function returns `true` iff p has a root in S. If the result is `true`, the function also returns a root of p in S as a second return value. Currently the coefficient rings that are allowed comprise complex and real fields, integers and rationals, finite fields and residue class rings with prime modulus. Note that particularly for finite fields, this method may be much faster than the computation of all roots of the polynomial.

`SmallRoots(p, N, X): RngUPolElt, RngElt, RngElt -> [RngElt]`

```
Bits       : BoolElt                  Default: false
Beta       : FldReElt                 Default: 1.0
Exponent   : RngIntElt                Default:
Finalshifts: RngIntElt                Default:
Direct     : BoolElt                  Default: false
```

Given a monic non-zero univariate integer polynomial p and two positive integers N and X, this function returns all x_0's such that $|x_0| \leq X$ and $P(x_0) = 0$ $[N]$, as long as $X \leq 0.5 \cdot N^{1/d}$, where d is the degree of p. This function implements Coppersmith's algorithm to compute the small roots of a univariate polynomial modulo an integer [Cop96], as described in Alexander May's PhD thesis [May03]. It relies upon the LLL algorithm for reducing euclidean lattices [LLL82]. It is frequently used for cryptanalysing public-key cryptosystems (see the example below).

When `Bits` is set to `true`, the input X is read as 2^X. The parameter `Beta` can be set to any value in $(0.0, 1.0]$. The routine will then find all x_0's such that $|x_0| \leq X$ and $P(x_0) = 0 \cong N'$, as long as $X \leq 0.5 \cdot N^{\beta^2/d}$, where d is the degree of p and $N' \geq N^\beta$ is any divisor of N. The `Exponent` and `Finalshifts` specify the shape of the lattice basis to be reduced. If `Exponent` is m, then p^m will be the highest power of p used to build the lattice basis, and if `Finalshifts` is t, t shifts of p^m will be used. Unless requested by the user, these parameters are chosen automatically. Finally, the `Direct` option allows the user to require the lattice basis to be reduced at once, and not progressively while constructed. This is can be slower.

Example: Factoring an RSA Modulus with Small Roots (ex-2984f7)

We show how to use the SmallRoots routine to factor an RSA modulus when some most significant bits of one of the factors is known. We first generate an RSA modulus.

```
> SetSeed(1);
> F<x> := PolynomialRing (Integers());
> length := 1024;
> p:=NextPrime (2^(Round(length/2)): Proof:=false);
> pi:=Pi(RealField());
> q:=NextPrime (Round (pi*p): Proof:=false);
> N := p*q;
```

Suppose that N is known, as well as an approximation of the factor q:

```
> hidden:=220;
> approxq := q+Random(2^hidden-1);
```

Our goal is to recover q from our knowledge of approxq. We are therefore interested in the small roots of the polynomial $x - approxq$ modulo q, whose multiple N is known.

```
> A:=x-approxq;
> time perturb:=SmallRoots (A, N, hidden : Bits, Beta:=0.5)[1];
Time 0.050
> q eq approxq-perturb;
true
```

`SetVerbose("SmallRoots", v): MonStgElt, RngIntElt`

(Procedure.) Set the verbose printing level for the SmallRoots routine to be v. Currently the legal values for v are true, false, 0, 1, or 2 (false is the same as 0, and true is the same as 1).

24.4.8 Derivative, Integral

`Derivative(p): RngUPolElt -> RngUPolElt`

Given a polynomial $p \in P$, return the derivative of p as an element of P.

`Derivative(p, n): RngUPolElt, RngIntElt -> RngUPolElt`

Given a polynomial $p \in P$ and an integer $n \geq 0$, return the n-th derivative of p as an element of P.

`Integral(p): RngUPolElt -> RngUPolElt`

Given a polynomial $p \in P$ over a field of characteristic zero, return the formal integral of p as an element of P.

24.4.9 Evaluation, Interpolation

`Evaluate(p, r): RngUPolElt, RngElt -> RngElt`

Given an element p of a polynomial ring P and an element r of a ring S, return the value of p evaluated at r. If r can be coerced into the coefficient ring R of P, the result will be an element of R. If r cannot be coerced to the coefficient ring, then an attempt is made to do a generic evaluation of p at r. In this case, the result will be an element of S.

`Interpolation(I, V): [ RngElt ], [ RngElt ] -> RngUPolElt`

This function finds a univariate polynomial that evaluates to the values V in the interpolation points I. Let K be a field and $n > 0$ an integer; given sequences I and V, both consisting of n elements of K, return the unique univariate polynomial p over K of degree less than n such that $p(I[i]) = V[i]$ for each $1 \leq i \leq n$.

24.4.10 Decomposition

The following function provides an inverse operation to composing polynomials.

`Decomposition(f): RngUPolElt -> [[RngUPolElt]]`

```
All: BoolElt                    Default: true
```

Given a polynomial 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.

Example: Polynomial Decomposition (ex-b0e795)

We show a decomposition of a polynomial and the corresponding evaluations which retrieve the polynomial which was decomposed.

```
> k:=GF(3^2);
> P<x>:=PolynomialRing(k);
> f:=x^9-x;
> Decomposition(f);
[
    [
        x^3 + x,
        x^3 + 2*x
    ],
    [
        x^3 + k.1^6*x,
        x^3 + k.1^6*x
    ],
    [
        x^3 + k.1^2*x,
        x^3 + k.1^2*x
    ],
    [
        x^3 + 2*x,
        x^3 + x
    ]
]
> Decomposition(f : All := false);
[
    [
        x^3 + x,
        x^3 + 2*x
    ]
]
> Evaluate($1[1][2], $1[1][1]) eq f;
true
```

24.4.11 Quotient and Remainder

`Quotrem(f, g): RngUPolElt, RngUPolElt -> RngUPolElt, RngUPolElt`

Given elements f and g of the polynomial ring $P = R[x]$, this function returns polynomials q (quotient) and r (remainder) in P such that $f = q \cdot g + r$, and the degree of r is minimal. The leading coefficient of g has to be a non-zero divisor in R. If the leading coefficient of g is a unit, then the degree of r will be strictly less than that of g (taking the degree of 0 to be -1). Over the integers ($R = \mathbb{Z}$) this will be true in general when the leading coefficient of g divides that of f.

`f div g: RngUPolElt, RngUPolElt -> RngUPolElt`

The quotient of the polynomial f by g, which is the first return value of `Quotrem` described above.

`IsDivisibleBy(f, g): RngUPolElt, RngUPolElt -> BoolElt, RngUPolElt`

Return whether the polynomial f is exactly divisible by the polynomial g; that is, whether there exists q with $f = qg$. If so, return also the exact divisor q.

`ExactQuotient(f, g): RngUPolElt, RngUPolElt -> RngUPolElt`

Assuming that the polynomial f is exactly divisible by the polynomial g, return the exact quotient of f by g (as a polynomial in the same polynomial ring). An error results if g does not divide f exactly.

`f mod g: RngUPolElt, RngUPolElt -> RngUPolElt`

The remainder of division of the polynomial f by g, which is the second return value of `Quotrem` described above.

`Valuation(f, g): RngUPolElt, RngUPolElt -> RngIntElt`

The exponent of the highest power of the polynomial g which divides the polynomial f.

`Reductum(f): RngUPolElt -> RngUPolElt`

The reductum of a polynomial f, which is the polynomial obtained by removing the leading term of f.

`PseudoRemainder(f, g): RngUPolElt, RngUPolElt -> RngUPolElt`

Given polynomials f, g in $P = R[x]$, where R is an integral domain, this function returns the pseudo-remainder r of f and g defined as follows. Let d be the maximum of 0 and $\deg(f) - \deg(g) + 1$, and let c be the leading coefficient of g; then r will be the unique polynomial in P such that $c^d \cdot f = q \cdot g + r$ and the degree of r is less than that of g (possibly -1 for $r = 0$).

`EuclideanNorm(p): RngUPol -> RngIntElt`

Return the Euclidean norm of the univariate polynomial $p \in P$, where the Euclidean norm is the function that makes P into a Euclidean ring, which is the degree function.

24.4.12 Modular Arithmetic

The following functions allow modular arithmetic for univariate polynomials over a field without the need to move into the quotient ring. See also the description of `mod` in the section on quotient and remainder.

`Modexp(f, n, g): RngUPolElt, RngIntElt, RngUPolElt -> RngUPolElt`

Given univariate polynomials f and g in $K[x]$ over a field K, return $f^n \bmod g$ as an element of $K[x]$. Here n must be a non-negative integer, and g is allowed to be a constant polynomial.

`ChineseRemainderTheorem(X, M): [RngUPolElt], [RngUPolElt] -> RngUPolElt`

`CRT(X, M): [RngUPolElt], [RngUPolElt] -> RngUPolElt`

Given two sequences X and M of polynomials where the elements in M are assumed to be pairwise coprime, find a single polynomial t that solve the modular equation $X_i = t$ modulo M_i.

24.4.13 Other Operations

`ReciprocalPolynomial(f): RngUPolElt -> RngUPolElt`

The reciprocal of the given univariate polynomial.

`PowerPolynomial(f, n): RngUPolElt, RngIntElt -> RngUPolElt`

The polynomial whose roots are the nth powers of the roots of the given polynomial (which should have coefficients in some field).

`f ^ M: RngUPolElt, Mtrx -> RngUPolElt`

The transformation of the univariate polynomial f under the linear fractional transformation given by the 2 by 2 matrix M (obtained by homogenizing f and making a linear substitution).

24.5 Common Divisors and Common Multiples

The functions in this section are restricted to univariate polynomials over a field, over the integers, or over a residue class ring of integers with prime modulus, or any polynomial ring over these.

24.5.1 Common Divisors and Common Multiples

`GreatestCommonDivisor(f, g): RngUPolElt, RngUPolElt -> RngUPolElt`

`Gcd(f, g): RngUPolElt, RngUPolElt -> RngUPolElt`

`GCD(f, g): RngUPolElt, RngUPolElt -> RngUPolElt`

Given univariate polynomials f and g over the ring R, this function returns the greatest common divisor (GCD) of f and g. The valid coefficient rings are those which themselves have a GCD algorithm for their elements (which includes most commutative rings in Magma).

If either of the inputs is zero, then the result is the other input (and if the inputs are both zero then the result is zero). The result is normalized (see the function *Normalize*), so the result is always unique.

For polynomials over finite fields, the simple Euclidean algorithm is used, since this is efficient (there is no intermediate coefficient blowup).

For polynomials over the integers or rationals, a combination of two algorithms is used: (1) the heuristic evaluation 'GCDHEU' algorithm of Char et al. ([CGG89] and [GCL92, section 7.7]), suitable for small to moderate-degree dense polynomials; (2) a modular algorithm similar to that presented in [vzGG99, Algorithm 6.38] or [GCL92, section 7.4] (although lifting all the way up to a bound is not used since it is completely unnecessary for correctness in this algorithm!).

For polynomials over an algebraic number field, quadratic field, or cyclotomic field, a fast modular algorithm is used, which maps the field to a residue class polynomial ring modulo a small prime.

Since V2.10, for polynomials over an algebraic function field or polynomial quotient ring over a function field, a new fast modular algorithm of Allan Steel (to be published) is used, which evaluates and interpolates for each base transcendental variable.

For polynomials over another polynomial ring or function field, the polynomials are first "flattened" to be inside a multivariate polynomial ring over the base coefficient ring, then the appropriate (multivariate) algorithm is used for that base coefficient ring.

For polynomials over any other ring, the generic subresultant algorithm [Coh93, section 3.3] is used.

`ExtendedGreatestCommonDivisor(f, g): RngUPolElt, RngUPolElt -> RngUPolElt, RngUPolElt, RngUPolElt`

`Xgcd(f, g): RngUPolElt, RngUPolElt -> RngUPolElt, RngUPolElt, RngUPolElt`

`XGCD(f, g): RngUPolElt, RngUPolElt -> RngUPolElt, RngUPolElt, RngUPolElt`

The extended greatest common divisor of polynomials f and g in a univariate polynomial ring P: the function returns polynomials c, a, and b in P with $\deg(a) < \deg(g)$ and $\deg(b) < \deg(f)$ such that c is the monic GCD of f and g, and $c = a \cdot f + b \cdot g$. The multipliers a and b are unique if f and g are both non-zero. The coefficient ring must be a field.

For polynomials over the rational field, a modular algorithm due to Allan Steel (unpublished) is used; over other fields the basic Euclidean algorithm is used.

`LeastCommonMultiple(f, g): RngUPolElt, RngUPolElt -> RngUPolElt`

`Lcm(f, g): RngUPolElt, RngUPolElt -> RngUPolElt`

`LCM(f, g): RngUPolElt, RngUPolElt -> RngUPolElt`

The least common multiple of polynomials f and g in a univariate polynomial ring P. The LCM of zero and anything else is zero. The result is normalized (see the function *Normalize*), so the result is always unique. The valid coefficient rings are as for the function `GCD`, above.

The LCM is effectively computed as `Normalize((F div GCD(F, G)) * G)`, for non-zero inputs.

`Normalize(f): RngUPolElt -> RngUPolElt`

Given a univariate polynomial f over the ring R, this function returns the unique normalized polynomial g which is associate to f (so $g = uf$ for some unit in R). This is chosen so that if R is a field then g is monic, if R is $\mathbb{Z}$ then the leading coefficient of g is positive, if R is a polynomial ring itself, then the leading coefficient of g is recursively normalized, and so on for other rings.

24.5.2 Content and Primitive Part

`Content(p): RngUPolElt -> RngIntElt`

The content of p, that is, the greatest common divisor of the coefficients of p as an element of the coefficient ring.

`PrimitivePart(p): RngUPolElt -> RngUPolElt`

The primitive part of p, being p divided by the content of p.

`ContentAndPrimitivePart(p): RngUPolElt -> RngIntElt, RngUPolElt`

`Contpp(p): RngUPolElt -> RngIntElt, RngUPolElt`

The content (the greatest common divisor of the coefficients) of p, as an element of the coefficient ring, as well as the primitive part (p divided by the content) of p.

24.6 Polynomials over the Integers

The functions in this section are available for univariate polynomials over the integers only.

`Sign(p): RngUPolElt -> RngIntElt`

The sign of the leading coefficient of p.

`AbsoluteValue(p): RngUPolElt -> RngUPolElt`

`Abs(p): RngUPolElt -> RngUPolElt`

Returns either p or $-p$ according to which one has non-negative leading coefficient.

`MaxNorm(p): RngUPolElt -> RngIntElt`

The maximum of the absolute values of the coefficients of p.

`SumNorm(p): RngUPolElt -> RngIntElt`

The sum of the coefficients of p.

`DedekindTest(p, m): RngUPolElt, RngIntElt -> Boolelt`

Given a monic polynomial p (univariate or multivariate in one variable) and a prime number m, this returns true if p satisfies the Dedekind criterion at m, and false otherwise. The Dedekind criterion is satisfied at m if and only if the equation order corresponding to p is locally maximal at m [PZ89, p. 295].

24.7 Polynomials over Finite Fields

The functions in this section are available for univariate polynomials over finite fields only.

`PrimePolynomials(R, d): RngUPol, RngIntElt -> SeqEnum[ RngUPolElt ]`

`PrimePolynomials(R, d, n): RngUPol, RngIntElt, RngIntElt -> SeqEnum[ RngUPolElt ]`

A sequence of all monic prime polynomials of R of degree d, resp. a sequence of n monic prime polynomials of R of degree d.

`RandomPrimePolynomial(R, d): RngUPol, RngIntElt -> RngUPolElt`

A random monic prime polynomial of R of degree d.

`NumberOfPrimePolynomials(q, d): RngIntElt, RngIntElt -> RngIntElt`

`NumberOfPrimePolynomials(K, d): FldFin, RngIntElt -> RngIntElt`

`NumberOfPrimePolynomials(R, d): RngUPol, RngIntElt -> RngIntElt`

The number of monic prime polynomials of degree d over the respective finite field.

`JacobiSymbol(a, b): RngUPol, RngUPol -> RngIntElt`

The Jacobi symbol (a/b) of the two polynomials $a, b \in \mathbb{F}_q[x]$ where q must be odd. If b is irreducible, the symbol equals 0 if b divides a. It equals 1 if a is a square mod b and -1 otherwise. The symbol then extends multiplicatively to all non-constant polynomials b.

24.8 Factorization

This section describes the functions for polynomial factorization and associated computations. These are available for several kinds of coefficient rings.

24.8.1 Factorization and Irreducibility

`Factorization(f): RngUPolElt -> [ < RngUPolElt, RngIntElt >], RngElt`
`Factorisation(f): RngUPolElt -> [ < RngUPolElt, RngIntElt >], RngElt`

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

Given a univariate polynomial f over the ring R, this function returns the factorization of f as a factorization sequence Q, that is, a sequence of pairs, each consisting of an irreducible factor q_i a positive integer k_i (its multiplicity). Each irreducible factor is normalized (see the function *Normalize*), so the expansion of the factorization sequence is the unique canonical associate of f. The function also returns the unit u of R giving the normalization, so $f = u \cdot \prod_i q_i^{k_i}$. The coefficient ring R must be one of the following: a finite field $\mathbb{F}_q$, the ring of integers $\mathbb{Z}$, the field of rationals $\mathbb{Q}$, an algebraic number field $\mathbb{Q}(\alpha)$, a local ring, or a polynomial ring, function field (rational or algebraic) or finite-dimensional affine algebra (which is a field) over any of the above. For factorization over very small finite fields, the Berlekamp algorithm is used by default, which depends on fast linear algebra (see, for example, [Knu97, section 4.6.2] or [vzGG99, section 14.8]). For medium to large finite fields, the von zur Gathen/Kaltofen/Shoup algorithm ([KS95, Sho95b, vzGS92]) is used by default. The parameter `Al` may be used to specify the factorization algorithm over finite fields. The possible values are:

(1) `"Default"`: The default strategy, whereby an appropriate choice will be made.

(2) `"BerlekampSmall"` or `"BerlekampLarge"` for the Berlekamp algorithm (see [Knu97, pp. 446--447] for the difference between these two variants).

(3) `"GKS"` for the von zur Gathen/Kaltofen/Shoup algorithm.

Since V2.8 (July 2001), Magma uses the algorithm of Mark van Hoeij [vH01, vH02] to factor polynomials over the integers or rationals. First a factorization of f is found modulo a suitable small prime, then Hensel lifting is applied, as in the standard Berlekamp-Zassenhaus (BZ) algorithm [Knu97, p. 452]. The Hensel lifting is performed using Victor Shoup's 'tree lifting' algorithm, as described in [vzGG99, Sec. 15.5]. Easy factors are also detected at various stages, if possible, using heuristics developed by Allan Steel. But the final search for the correct combination of modular factors (which has exponential worst-case complexity in the standard BZ algorithm) is now performed by van Hoeij's algorithm, which efficiently finds the correct combinations by solving a Knapsack problem via the LLL lattice-basis reduction algorithm [LLL82].

van Hoeij's new algorithm is *much* more efficient in practice than the original lattice-based factoring algorithm proposed in [LLL82]: the lattice constructed in van Hoeij's algorithm has dimension equal to the number of modular factors (not the degree of the input polynomial), and the entries of the lattice are very much smaller. Many polynomials can now be easily factored which were out of reach for any previous algorithm (see the examples below).

For polynomials over algebraic number fields, algebraic function fields, and affine algebras, the norm-based algorithm of Trager [Tra76] is used, which performs a suitable substitution and resultant computation, and then factors the resulting polynomial with one less variable. In characteristic

zero, the difficult case (where there are very many factors of this integral polynomial modulo any prime) is now easily handled by van Hoeij's combination algorithm above. In small characteristic, where inseparable field extensions may occur, an algorithm of Allan Steel ([Ste05]) is used.

`HasPolynomialFactorization(R): Rng -> BoolElt`

Given a ring R, return whether factorization of polynomials over R is allowed in Magma.

`SetVerbose("PolyFact", v): MonStgElt, RngIntElt`

(Procedure.) Change the verbose printing level for all polynomial factorization algorithms to be v. Currently the legal levels are 0, 1, 2, or 3.

`FactorisationToPolynomial(f): [Tup] -> BoolElt`

`Facpol(f): [Tup] -> BoolElt`

Given a sequence of tuples, each consisting of pairs of irreducible polynomials and positive integer exponents, return the product polynomial.

Example: Factoring Swinnerton-Dyer Polynomials `(ex-d3c652)`

To demonstrate the power of the van Hoeij combination algorithm, in this example we factor Swinnerton-Dyer polynomials, which are worse-case inputs for the Berlekamp-Zassenhaus factorization algorithm for polynomials over $\mathbb{Z}$.

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. This polynomial lies in $\mathbb{Z}[x]$, has degree 2^n, is irreducible over $\mathbb{Z}$, and has at least 2^{n-1} factors modulo any prime. This last fact is easy to see, since, given any finite field K, the polynomial must split into linear factors over a quadratic extension of K, so it will have only linear or quadratic factors over K. See also [vzGG99, section 15.3] for further discussion.

In this example, we use the function `SwinnertonDyerPolynomial` to construct the polynomials (see Example `Swinnerton-Dyer Polynomials (ex-681d9c)` in the chapter on algebraically closed fields for an explanation of how this function works).

First we display the first 4 polynomials.

```
> P<x> := PolynomialRing(IntegerRing());
> SwinnertonDyerPolynomial(1);
x^2 - 2
> SwinnertonDyerPolynomial(2);
x^4 - 10*x^2 + 1
> SwinnertonDyerPolynomial(3);
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
> SwinnertonDyerPolynomial(4);
x^16 - 136*x^14 + 6476*x^12 - 141912*x^10 + 1513334*x^8 - 7453176*x^6 +
    13950764*x^4 - 5596840*x^2 + 46225
> IsIrreducible($1);
true
```

We note the degree patterns of the factorizations of the first eight Swinnerton-Dyer polynomials over the three finite fields $\mathbf{F}_3$, $\mathbf{F}_{23}$, and $\mathbf{F}_{503}$. There are only linear or quadratic factors, as expected.

```
> for i := 1 to 8 do
>    f := SwinnertonDyerPolynomial(i);
>    printf "%o:", i;
>    for p in [3, 23, 503] do
>        L := Factorization(PolynomialRing(GF(p)) ! f);
>        printf " %o", {* Degree(t[1])^^t[2]: t in L *};
>     end for;
```

(continues on next page)

(continued from previous page)

```
>     "";
> end for;
1: {* 2 *} {* 1^^2 *} {* 1^^2 *}
2: {* 2^^2 *} {* 1^^4 *} {* 1^^4 *}
3: {* 1^^4, 2^^2 *} {* 2^^4 *} {* 2^^4 *}
4: {* 1^^8, 2^^4 *} {* 2^^8 *} {* 2^^8 *}
5: {* 1^^8, 2^^12 *} {* 2^^16 *} {* 2^^16 *}
6: {* 1^^16, 2^^24 *} {* 2^^32 *} {* 2^^32 *}
7: {* 1^^48, 2^^40 *} {* 2^^64 *} {* 2^^64 *}
8: {* 1^^96, 2^^80 *} {* 1^^16, 2^^120 *} {* 2^^128 *}
```

We now construct the 6-th polynomial, note its largest coefficient, and then factor it; it takes only a second to prove that it is irreducible, even though there are 32 modular factors.

```
> sd6 := SwinnertonDyerPolynomial(6);
> Degree(sd6);
64
> Max([Abs(x): x in Coefficients(sd6)]);
177108072043062916168515897889215259945611
> time L := Factorization(sd6);
Time: 1.009
> #L;
1
```

Now we factor the 7-th polynomial!

```
> sd7 := SwinnertonDyerPolynomial(7);
> Degree(sd7);
128
> Max([Abs(x): x in Coefficients(sd7)]);
85783447140360187781662744163364252674665633803596496806969245871\\
44011458425706833248256 19
> time L := Factorization(sd7);
Time: 11.670
> #L;
1
```

We now factor the product of the 6-th and 7-th polynomials. This has degree 192 and has at least 96 factors modulo any prime! But the van Hoeij algorithms easily finds the correct factors over the integers.

```
> p := sd6*sd7;
> Degree(p);
192
> Max([Abs(x): x in Coefficients(p)]);
46178075233031441597519883536198372339486796800578859978206259791\\
48178917111255021010981707011266628489195528524859249200516300831
> time L := Factorization(p);
Time: 16.840
> #L;
2
> L[1,1] eq sd6;
true
> L[2,1] eq sd7;
true
```

See also Example Swinnerton-Dyer Polynomials (ex-681d9c) in the chapter on algebraically

closed fields for a generalization of the Swinnerton-Dyer polynomials.

`SquarefreeFactorization(f): RngUPolElt -> [ <RngUPolElt, RngIntElt> ]`

Given a univariate polynomial f over the ring R, this function returns the squarefree factorization of f as a sequence of pairs, each consisting of a (not necessarily irreducible) factor and an integer indicating the multiplicity. The factors do not contain the square of any non-constant polynomial.

The coefficient ring R must be the integer ring or any field. The algorithm works by computing the GCD of f with its derivative and repeating as necessary (special considerations are also necessary for characteristic p).

`DistinctDegreeFactorization(f): RngUPolElt -> [ <RngIntElt, RngUPolElt> ]`

`Degree: RngIntElt`	`Default: 0`

Given a squarefree univariate polynomial $f \in F[x]$ with F a finite field, this function returns the distinct-degree factorization of f as a sequence of pairs, each consisting of a degree d, together with the product of the degree-d irreducible factors of f. If the optional parameter `Degree` is given a value $L > 0$, then only (products of) factors up to degree L are returned.

`EqualDegreeFactorization(f, d, g): RngUPolElt, RngIntElt, RngUPolElt -> [ RngUPolElt ]`

Given a squarefree univariate polynomial $f \in F[x]$ with F a finite field, and integer d and another polynomial $g \in F[x]$ such that F is known to be the product of distinct degree-d irreducible polynomials alone, and g is $x^q \bmod f$, where q is the cardinality of F, this function returns the irreducible factors of f as a sequence of polynomials (no multiplicities are needed). If the conditions are not satisfied, the result is unpredictable. This function allows one to split f, assuming that one has computed f in some special way.

`IsIrreducible(f): RngUPolElt -> BoolElt`

Given a univariate polynomial f over the ring R, this function returns `true` if and only f is irreducible over R. The conditions on R are the same as for the function *Factorization* above.

`IsSeparable(f): RngUPolElt -> BoolElt`

Given a polynomial $f \in K[x]$ such that f is a polynomial of degree ≥ 1 and K is a field allowing polynomial factorization, this function returns `true` iff f is separable.

`QMatrix(f): RngUPolElt -> AlgMatElt`

Given a univariate polynomial f of degree d over a finite field F this function returns the Berlekamp Q-matrix associated with f, which is an element of the degree $d-1$ matrix algebra over F.

24.8.2 Sylvester Matrix, Resultant, and Discriminant

`SylvesterMatrix(f, g): RngUPolElt, RngUPolElt -> Mtrx`

The Sylvester matrix of univariate polynomials f and g (of degree m and n respectively). This is an $(m+n) \times (m+n)$ matrix with entries in R, which is constructed by first creating n rows from the coefficients of f, followed by m rows from the coefficients of g. Within the two above blocks for rows, each subsequent row is a shifted version of the row above, with zeros filling in the empty spaces. The coefficient ring R may be any ring.

`Resultant(f, g): RngUPolElt, RngUPolElt -> RngElt`

The resultant of univariate polynomials f and g (of degree m and n respectively) in $R[x]$, which is by definition the determinant of the Sylvester matrix of f and g. The resultant is an element of R. The coefficient ring R may be any commutative ring.

`Discriminant(f): RngUPolElt -> RngIntElt`

The discriminant D of $f \in R[x]$, where R is any commutative ring. This is an element of R that is exactly equivalent to the Magma call:

```
LeadingCoefficient(f)^(Degree(f) - Degree(d) - 2) * Resultant(f, d)
```

where `d := Derivative(f)`.

`CompanionMatrix(f): RngUPolElt -> AlgMatElt`

Given a monic univariate polynomial f of degree d over some ring R, return the companion matrix of f as an element of the full matrix algebra of degree $d-1$ over R. The companion matrix for $f = a_0 + a_1 x + \cdots + a_{d-1}x^{d-1} + x^d$ is given by

$$\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ -a_0 & -a_1 & -a_2 & \cdots - a_{d-2} & -a_{d-1} \end{pmatrix}.$$

24.8.3 Hensel Lifting

`HenselLift(f, s, P): RngUPolElt, [ RngUPolElt ], RngUPol -> [ RngUPolElt ]`

Given the sequence of irreducible factors s modulo some prime p of the univariate integer polynomial f, return the Hensel lifting into the polynomial ring P, which must be the univariate polynomial ring over a residue class ring modulo some power of p. Thus given $f \equiv \prod_i s_i \bmod p$, this returns $f \equiv \prod_i t_i \bmod p^k$ for some $k \geq 1$, as a sequence of polynomials in $\mathbb{Z}/p^k\mathbb{Z}$. The factorization of f modulo p must be squarefree, that is, s should not contain repeated factors.

Example: Hensel Lifting (ex-4d71ca)

```
> R<x> := PolynomialRing(Integers());
> b := x^5 - x^3 + 2*x^2 - 2;
> F<f> := PolynomialRing(GF(5));
> s := [ w[1] : w in Factorization( F ! b ) ];
> s;
[
    f + 1,
    f + 3,
    f + 4,
    f^2 + 2*f + 4
]
> T<t> := PolynomialRing(Integers(5^3));
> h := HenselLift(b, s, T);
> h;
[
    t + 1,
    t + 53,
    t + 124,
    t^2 + 72*t + 59
]
> &*h;
t^5 + 124*t^3 + 2*t^2 + 123
```

24.9 Ideals and Quotient Rings

Currently it is only possible to create ideals and quotient rings in univariate polynomial rings over fields. Note that these are principal ideal domains: all ideals can be generated by a single element.

24.9.1 Creation of Ideals and Quotients

`ideal< R | a1, ..., ar >: RngUPol, RngUPolElt, ..., RngUPolElt -> RngUPol`

Given a univariate polynomial ring R over a field K, this function returns the ideal of R generated by the elements $a_1, \ldots, a_r \in R$. This is the same as the ideal generated by the greatest common divisor of the elements a_i in R. The function returns the ideal as a subring of R, generated by a single element.

`quo< R | I >: RngUPol, RngUPol -> RngUPolRes`

`quo< R | a`$_1$`, ..., a`$_r$` >: RngUPol, RngUPolElt, ..., RngUPolElt -> RngUPolRes, Map`

Given an ideal I in the univariate polynomial ring R (over a field), return the quotient R/I, as well as the projection map $h : R \to R/I$. The ideal I may either be specified as an ideal or by a list a_1, a_2, ..., a_r, of generators. The angle bracket notation can be used to assign names to the indeterminates: `Q<q> := quo< I | I >;`.

24.9.2 Ideal Arithmetic

Since ideals of R are regarded as subrings of R, the ring R itself is a valid ideal as well.

`I + J: RngUPol, RngUPol -> RngUPol`

Given ideals I and J in the same polynomial ring R, this function returns the sum of the ideals I and J, which is the ideal generated by the generators of I and those of J. Since we require R to be a principal ideal domain, the resulting ideal will be simply generated by the greatest common divisor of `I.1` and `J.1`.

`I * J: RngUPol, RngUPol -> RngUPol`

Given ideals I and J in the same polynomial ring R, this function returns the product of the ideals I and J, which is the ideal generated by the products of the generators of I and those of J. Since we require R to be a principal ideal domain, the resulting ideal will be simply generated by `I.1 * J.1`.

`I meet J: RngUPol, RngUPol -> RngUPol`

Given ideals I and J in the same polynomial ring R, this function returns the intersection of the ideals I and J. Since we require R to be a principal ideal domain, the resulting ideal will equal the product of I and J and be simply generated by `I.1 * J.1`.

`a in I: RngUPolElt, RngUPol -> BoolElt`

Given an element a of a polynomial ring P as well as an ideal I of P, this function returns `true` if and only if a is contained in I, and `false` otherwise.

`a notin I: RngUPolElt, RngUPol -> BoolElt`

Given an element a of a polynomial ring P as well as an ideal I of P, this function returns `false` if and only if a is contained in I, and `true` otherwise.

`I eq J: RngUPol, RngUPol -> BoolElt`

Given two ideals I and J in the same polynomial ring R this returns `true` if and only if I and J are the same, and `false` otherwise.

`I ne J: RngUPol, RngUPol -> BoolElt`

Given two ideals I and J in the same polynomial ring R this returns `false` if and only if I and J are the same, and `true` otherwise.

`I subset J: RngUPol, RngUPol -> BoolElt`

Given two ideals I and J in the same polynomial ring R this returns `true` if and only if I is contained in J, and `false` otherwise.

`I notsubset J: RngUPol, RngUPol -> BoolElt`

Given two ideals I and J in the same polynomial ring R this returns `false` if and only if I is contained in J, and `true` otherwise.

24.9.3 Other Functions on Ideals

Since ideals are considered as subrings of polynomial rings, and in particular are in the same Magma category as polynomial rings, most of the function listed in this chapter for polynomial rings do also apply to ideals, but some restrictions apply. Thus it will be possible to get the coefficient ring but it will not be possible to use `ChangeRing` to change it. We list some functions here that additional comments.

`I . 1: RngUPol -> RngUPolElt`

Given an ideal I in a univariate polynomial ring R, return the generator of I in R as an element of I.

24.9.4 Other Functions on Quotients

Contrary to ideals, quotient rings form a separate Magma category. Only very few functions are available on these rings; however most element functions for polynomial rings apply to elements of quotients as well, in particular the coefficient, term, and degree functions.

`Modulus(Q): RngUPolRes -> RngUPolElt`

Given a quotient ring $Q = R[x]/I$ of the univariate polynomial ring $R[x]$ obtained by factoring out by the ideal I, return the generator for I as an element of R.

`PreimageRing(Q): RngUPolRes -> RngUPol`

If Q is the quotient $Q = R/I$ for some univariate polynomial ring R, this function returns R.

24.10 Special Families of Polynomials

24.10.1 Orthogonal Polynomials

`ChebyshevFirst(n): RngIntElt -> RngUPolElt`

`ChebyshevT(n): RngIntElt -> RngUPolElt`

Given a positive integer n, this function constructs the Chebyshev polynomial of the first kind $T_n(x)$, where $T_n(x)$ is defined by $T_n(x) = \cos n\theta$ with $x = \cos\theta$.

`ChebyshevSecond(n): RngIntElt -> RngUPolElt`

`ChebyshevU(n): RngIntElt -> RngUPolElt`

Given a positive integer n, this function constructs the Chebyshev polynomial of the second kind, $U_n(x)$, of degree $n-1$. The polynomial is defined by $U_n(x) = \frac{1}{n}T'_n(x) = \frac{\sin n\theta}{\sin\theta}$ where $x = \cos\theta$.

`LegendrePolynomial(n): RngIntElt -> RngUPolElt`

Given a positive integer n, this function constructs the Legendre polynomial $P_n(x)$ of degree n, where $P_n(x)$ is defined by

$$P_0(x) = 1,\ P_1(x) = x,$$
$$P_n(x) = \frac{1}{n}((2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)).$$

`LaguerrePolynomial(n): RngIntElt -> RngUPolElt`

`LaguerrePolynomial(n, m): RngIntElt, RngElt -> RngUPolElt`

Given a positive integer n, this function constructs the Laguerre polynomial $L_n^m(x)$ of degree n with parameter m. If m is omitted, it is assumed to be zero if it is not specified. The polynomial satisfies the recurrence relation

$$L_0(x) = 1,\ L_1(x) = 1 + m - x,$$
$$L_n(x) = \frac{1}{n}(((2n+m-1)-x)L_{n-1}^m(x) - (n-1+m)L_{n-2}^m(x)).$$

`HermitePolynomial(n): RngIntElt -> RngUPolElt`

Given a positive integer n, this function constructs the Hermite polynomial $H_n(x)$ of degree n, where $H_n(x)$ is defined by

$$H_0(x) = 1,\ H_1(x) = 2x,$$
$$H_n(x) = 2xH_{n-1}(x) - 2nH_{n-2}(x).$$

`GegenbauerPolynomial(n, m): RngIntElt, RngElt -> RngUPolElt`

Given a positive integer n and an integer m, this function constructs the Gegenbauer polynomial $C_n^m(x)$ of degree n with parameter m, where $C_n^m(x)$ is defined by

$$C_0^m(x) = 1,\ C_1^m(x) = 2mx,$$
$$C_n^m(x) = \frac{1}{n}(2(n-1+m)xC_{n-1}^m(x) - (n+2m-2)C_{n-2}^m(x)).$$

24.10.2 Permutation Polynomials

`DicksonFirst(n, a): RngIntElt, RngElt -> RngUPolElt`

Given a positive integer n, this function constructs the Dickson polynomial of the first kind $D_n(x, a)$ of degree n, where $D_n(x, a)$ is defined by

$$D_n(x,a) = \sum_{i=0}^{\lfloor n/2 \rfloor} \frac{n}{n-i}\binom{n-i}{i}(-a)^i x^{n-2i}.$$

`DicksonSecond(n, a): RngIntElt, RngElt -> RngUPolElt`

Given a positive integer n, this function constructs the Dickson polynomial of the second kind $E_n(x, a)$ of degree n, where $E_n(x, a)$ is defined by

$$E_n(x,a) = \sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n-i}{i}(-a)^i x^{n-2i}.$$

24.10.3 The Bernoulli Polynomial

`BernoulliPolynomial(n): RngIntElt -> RngUPolElt`

Given a positive integer n, this function constructs the n-th Bernoulli polynomial.

24.10.4 Swinnerton-Dyer Polynomials

`SwinnertonDyerPolynomial(n): RngIntElt -> RngUPolElt`

Given a positive integer n, this function constructs the n-th Swinnerton-Dyer polynomial, which 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. This polynomial lies in $\mathbb{Z}[x]$, has degree 2^n, and is irreducible over $\mathbb{Z}$.

See Example *Factoring Swinnerton-Dyer Polynomials* above which explains more about this class of polynomials, and see also Example `Swinnerton-Dyer Polynomials (ex-681d9c)` in the chapter on algebraically closed fields to see how these polynomials are constructed and also for a generalization.

24.11 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.
- [CGG89] Bruce W. Char, Keith O. Geddes, and Gaston H. Gonnet. GCDHEU: Heuristic Polynomial GCD Algorithm Based on Integer GCD Computation. *J. Symbolic Comp.*, 7(1):31–48, 1989.
- [Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory.* Volume 138 of Graduate Texts in Mathematics. Springer, Berlin–Heidelberg–New York, 1993.

- [Cop96] Don Coppersmith. Finding a small root of a univariate modular equation. In *Advances in Cryptology—EuroCrypt 1996*, volume 1070 of LNCS, pages 155–165. Springer, 1996.
- [GCL92] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Kluwer, Boston/Dordrecht/London, 1992.
- [KS95] Erich Kaltofen and Victor Shoup. Subquadratic-time factoring of polynomials over finite fields. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, 398–406. ACM, 1995.
- [Knu97] Donald E. Knuth. *The Art of Computer Programming*. Volume 2. Addison Wesley, Reading, Massachusetts, 3rd edition, 1997.
- [LLL82] Arjen K. Lenstra, Hendrik W. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.
- [May03] Alexander May. *New RSA Vulnerabilities Using Lattice Reduction Methods*. Dissertation, University of Paderborn, 2003.
- [PZ89] Michael E. Pohst and Hans Zassenhaus. *Algorithmic Algebraic Number Theory*. Encyclopaedia of mathematics and its applications. Cambridge University Press, Cambridge, 1989.
- [Sho95b] Victor Shoup. A New Polynomial Factorization Algorithm and its Implementation. *J. Symbolic Comp.*, 20(4):363–397, 1995.
- [Ste05] Allan Steel. Conquering Inseparability: Primary Decomposition and Multivariate Factorization over Algebraic Function Fields of Positive Characteristic. *J. Symbolic Comp.*, 40(3):1053–1075, 2005.
- [Tra76] Barry M. Trager. Algebraic factoring and rational function integration. In R.D. Jenks, editor, *Proc. SYMSAC '76*, 196–208. ACM press, 1976.
- [vH01] Mark van Hoeij. Factoring Polynomials and 0-1 vectors. In *Proceedings of the Cryptography and Lattices Conference (CaLC 2001), Brown University, Providence, RI, USA, March 29-30, 2001*, 142–146. Springer, 2001.
- [vH02] Mark van Hoeij. Factoring Polynomials and the knapsack problem. *J. Number Th.*, 95(2):167–189, 2002. http://www.math.fsu.edu/∼hoeij/paper/knapsack.ps.
- [vzGG99] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, 1999.
- [vzGS92] Joachim von zur Gathen and Victor Shoup. Computing Frobenius Maps And Factoring Polynomials. *Computational Complexity*, 2:187–224, 1992.

Chapter 25

Multivariate Polynomial Rings

25.1 Introduction

This chapter describes multivariate polynomial rings in Magma. A multivariate polynomial ring in any number of variables $n \geq 1$ can be created over an arbitrary coefficient ring R, and we will denote it by $P = R[x_1, \ldots, x_n]$. Certain functions, however, will only apply for coefficient rings satisfying certain conditions. Magma contains a powerful system for computing with ideals of multivariate polynomial rings. This is based on the construction of Gröbner bases of such ideals. This chapter only deals with polynomial rings and operations on their elements; see Chapter Gröbner Bases for the details concerning ideals and Gröbner bases. Permutation and matrix groups have a natural action on multivariate polynomial rings. This leads to the subject of invariant rings of finite groups, which is covered in Chapter Invariant Theory. See also the chapters on affine algebras (Chapter Affine Algebras) and on modules over affine algebras (Chapter Modules over Multivariate Rings).

25.1.1 Representation

Let P be the polynomial ring $R[x_1, \ldots, x_n]$ of rank n over a ring R. A *monomial* (or *power product*) of P is a product of powers of the variables of P, that is, an expression of the form $x_1^{e_1} \cdots x_n^{e_n}$ with $e_i \geq 0$ for $1 \leq i \leq n$. Multivariate polynomials in Magma are stored efficiently in distributive form, using arrays of coefficient-monomial pairs, where the coefficient is in the base ring R. The word 'term' will always refer to a coefficient multiplied by a monomial. Various orders can be applied to the monomials, and these are of great importance when dealing with Gröbner bases. A polynomial ring in Magma may defined with a certain monomial order, but as this does not affect the basic arithmetic operations in the polynomial ring, these orders are not described here but in the chapter dealing with Gröbner bases (see Section Representation and Monomial Orders). Since V2.7 (June 2000), a new generalized monomial representation has been developed, which uses differing byte sizes for monomials depending on the size of the monomials encountered. Monomial overflow is rigorously detected, and the system automatically extends the byte size of the monomials in the background if possible. Thus there is no need for the user to know beforehand the maximum degree which may occur, and much memory is also saved for low- to medium-degree computations. The total degree of any monomial may now be anything up to $2^{30} - 1 = 1073741823$. It is possible but not advised to use distributive 'multivariate' polynomials in one single variable. (See Chapter *Univariate Polynomial Rings* which is devoted to univariate polynomial rings.)

25.2 Polynomial Rings and Polynomials

25.2.1 Creation of Polynomial Rings

Multivariate polynomial rings are created from a coefficient ring, the number of indeterminates, and a monomial order. If no order is specified, the monomial order is taken to be the lexicographical order (see Section Representation and Monomial Orders for details).

`PolynomialRing(R, n): Rng, RngIntElt -> RngMPol`

`PolynomialAlgebra(R, n): Rng, RngIntElt -> RngMPol`

```
Global: BoolElt                    Default: false
```

Create a multivariate polynomial ring in $n > 0$ indeterminates over the ring R. The ring is regarded as an R-algebra via the usual identification of elements of R and the constant polynomials. The lexicographical ordering on the monomials is used for this default construction (see Section Representation and Monomial Orders). The angle bracket notation can be used to assign names to the indeterminates; e.g.: `P<x, y> := PolynomialRing(R, 2);` etc. By default, a *non-global* polynomial ring will be returned; if the parameter `Global` is set to `true`, then the unique global polynomial ring 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 rings with the same numbers of variables but with different variable names (and create mappings between them, for example). Explicit coercion is always allowed between polynomial rings 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 ring to the i-th variable of the other ring.

`PolynomialRing(R, n, order): Rng, RngIntElt, MonStgElt, ... -> RngMPol`

`PolynomialAlgebra(R, n, order): Rng, RngIntElt, MonStgElt, ... -> RngMPol`

Create a multivariate polynomial ring in $n > 0$ indeterminates over the ring R with the given order order on the monomials. See Section Representation and Monomial Orders for details.

Example: Assigning Generator Names `(ex-1b3506)`

We show the use of angle brackets for generator names.

```
> Z := IntegerRing();
> S := PolynomialRing(Z, 2);
```

If we define S this way, we can only refer to the indeterminates by $S.1$ and $S.2$ (see below). So we could assign these generators to variables, say x and y, as follows:

```
> x := S.1;
> y := S.2;
```

In this case it is easy to construct polynomials, but printing is slightly awkward:

```
> f := x^3*y +3*y^2;
> f;
$.1^3*$.2 + 3*$.2^2
```

To overcome that, it is possible to assign names to the indeterminates that are used in the printing routines, using the `AssignNames` function, before assigning to x and y.

```
> AssignNames(~S, ["x", "y"]);
> x := S.1; y := S.2;
> f := x^3*y +3*y^2;
> f;
x^3*y + 3*y^2
```

Alternatively, we use the angle brackets to assign generator names that will be used in printing as well:

```
> S<x, y> := PolynomialRing(Z, 2);
> f := x^3*y +3*y^2;
> f;
x^3*y + 3*y^2
```

Example: Global and Non-Global Polynomial Rings (ex-a068bd)

We demonstrate the difference between global and non-global rings. We first create the global multivariate polynomial ring over $\mathbb{Q}$ with 3 variables twice.

```
> Q := RationalField();
> P<x,y,z> := PolynomialRing(Q, 3: Global);
> PP := PolynomialRing(Q, 3: Global);
> P;
Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
> PP;
Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
> PP.1;
x
```

PP is identical to P. We now create default (non-global) multivariate polynomial rings (which are also different to the global polynomial ring P). Explicit coercion is allowed, and maps the i-variable of one ring to the i-th variable of the other ring.

```
> P1<a,b,c> := PolynomialRing(Q, 3);
> P2<d,e,f> := PolynomialRing(Q, 3);
> P1;
Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: a, b, c
> P2;
Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: d, e, f
> a;
a
> d;
d
> P1 ! d;
a
> P ! e;
y
```

25.2.2 Print Names

The `AssignNames` and `Name` functions can be used to associate names with the indeterminates of polynomial rings after creation.

`AssignNames(~P, s): RngMPol, [ MonStgElt ]`

Procedure to change the name of the indeterminates of a polynomial ring P. 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 P, in which case the remaining indeterminate names remain unchanged. This procedure only changes the name used in printing the elements of P. It does *not* assign to identifiers corresponding to the strings the indeterminates in P; to do this, use an assignment statement, or use angle brackets when creating the polynomial ring. Note that since this is a procedure that modifies P, it is necessary to have a reference `~P` to P in the call to this function.

`Name(P, i): RngMPol, RngIntElt -> RngMPolElt`

Given a polynomial ring P, return the i-th indeterminate of P (as an element of P).

25.2.3 Graded Polynomial Rings

It is possible within Magma to assign weights to the variables of a multivariate polynomial ring. This means that monomials of the ring then have a *weighted degree* with respect to the weights of the variables. Such a multivariate polynomial ring is called *graded* or *weighted*. Since this subject is intimately related to ideals, it is covered in the chapter on ideals and Gröbner bases (see Subsection Creation of Graded Polynomial Rings).

25.2.4 Creation of Polynomials

The easiest way to create polynomials in a given ring is to use the angle bracket construction to attach variables to the indeterminates, and then to use these variables to create polynomials (see the examples). Below we list other options.

`P . i: RngMPol, RngInt -> RngMPolElt`

Return the i-th indeterminate for the polynomial ring P in n variables ($1 \leq i \leq n$) as an element of P.

`elt< R | a >: RngMPol, RngElt -> RngMPolElt`

`R ! s: RngMPol, RngElt -> RngMPolElt`

`R ! s: RngMPol, [ RngElt ] -> RngMPolElt`

`elt< R | s >: RngMPol, [ RngElt ] -> RngMPolElt`

This element constructor can only be used for trivial purposes in multivariate polynomial rings: given a polynomial ring $P = R[x_1, \ldots, x_n]$ and an element a that can be coerced into the coefficient ring R, the constant polynomial a is returned; if a is in P already it will be returned unchanged.

`MultivariatePolynomial(P, f, i): RngMPol, RngUPolElt, RngIntElt -> RngMPolElt`

`MultivariatePolynomial(P, f, v): RngMPol, RngUPolElt, RngMPolElt -> RngMPolElt`

Given a multivariate polynomial ring $P = R[x_1, \ldots, x_n]$, as well as a polynomial f in a univariate polynomial ring $R[x]$ over the same coefficient ring R, return an element q of P corresponding to f in the indeterminate $v = x_i$; that is, $q \in P$ is defined by $q = \sum_j f_j x_i^j$ where $f = \sum_j f_j x^j$. The indeterminate x_i can either be specified as a polynomial $v = x_i$ in P, or by simply providing the integer i with $1 \leq i \leq n$. The inverse operation is performed by the `UnivariatePolynomial` function.

`One(P): RngMPol -> RngMPolElt`

`Identity(P): RngMPol -> RngMPolElt`

`Zero(P): RngMPol -> RngMPolElt`

`Representative(P): RngMPol -> RngMPolElt`

25.3 Structure Operations

25.3.1 Related Structures

The main structure related to a polynomial ring is its coefficient ring. Multivariate polynomial rings belong to the Magma category `RngMPol`.

`BaseRing(P): RngMPol -> Rng`

`CoefficientRing(P): RngMPol -> Rng`

Return the coefficient ring of polynomial ring P.

`Category(P): RngMPol -> Cat`

`Parent(P): RngMPol -> Pow`

`PrimeRing(P): RngMPol -> Rng`

25.3.2 Numerical Invariants

Note that the # operator only returns a value for finite (quotients of) polynomial rings.

`Rank(P): RngMPol -> RngIntElt`

Return the number of indeterminates of polynomial ring P over its coefficient ring.

`Characteristic(P): RngMPol -> RngIntElt`

`#P: RngMPol -> RngIntElt`

25.3.3 Ring Predicates and Booleans

The usual ring functions returning Boolean values are available on polynomial rings.

`IsCommutative(P): RngMPol -> BoolElt`

`IsUnitary(P): RngMPol -> BoolElt`

`IsFinite(P): RngMPol -> BoolElt`

`IsOrdered(P): RngMPol -> BoolElt`

`IsField(P): RngMPol -> BoolElt`

`IsEuclideanDomain(P): RngMPol -> BoolElt`

`IsPID(P): RngMPol -> BoolElt`

`IsUFD(P): RngMPol -> BoolElt`

`IsDivisionRing(P): RngMPol -> BoolElt`

`IsEuclideanRing(P): RngMPol -> BoolElt`

`IsDomain(P): RngMPol -> BoolElt`

`IsPrincipalIdealRing(P): RngMPol -> BoolElt`

`P eq Q: RngMPol, RngMPol -> BoolElt`

`P ne Q: RngMPol, RngMPol -> BoolElt`

25.3.4 Changing Coefficient Ring

The `ChangeRing` function enables the changing of the coefficient ring of a polynomial ring.

`ChangeRing(P, S): RngMPol, Rng -> RngMPol`

Given a polynomial ring $P = R[x_1, \ldots, x_n]$ of rank n with coefficient ring R, together with a ring S, construct the polynomial ring $Q = S[x_1, \ldots, x_n]$. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S.

25.3.5 Homomorphisms

In its general form, a ring homomorphism taking a polynomial ring $R[x_1, \ldots, x_n]$ 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 >: RngMPol, Rng -> Map`

`hom< P -> S | y1, ..., yn >: RngMPol, Rng -> Map`

Given a polynomial ring $P = R[x_1, \ldots, x_n]$, a ring S, a map $f : R \to S$, and n elements $y_1, \ldots, y_n \in S$, create the homomorphism $g : P \to S$ by applying the rules that $g(rx_1^{a_1} \cdots x_n^{a_n}) = f(r)y_1^{a_1} \cdots y_n^{a_n}$ for monomials and linearity, that is, $g(M + N) = g(M) + g(N)$. 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 into a Number Field (ex-a59b0c)

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();
> R<x, y> := PolynomialRing(Q, 2);
> A<a> := PolynomialRing(IntegerRing());
> N<z, w> := NumberField([a^3-2, a^2+5]);
> h := hom< R -> N | z, w >;
> h(x^11*y^3-x+4/5*y-13/4);
-40*w*z^2 - z + 4/5*w - 13/4
```

25.4 Element Operations

25.4.1 Arithmetic Operators

The usual unary and binary ring operations are available for multivariate polynomials. For polynomial rings over fields division by elements of the coefficient field are allowed (with the result in the original polynomial ring). The operator `div` has slightly different semantics from the univariate case: if b divides a, that is, if there exists a polynomial $q \in P$ such that $a = b \cdot q \in P$ then q will be the result of `a div b`, but if such polynomial does not exist an error results.

`+ a: RngMPolElt -> RngMPolElt`

`- a: RngMPolElt -> RngMPolElt`

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

`a - b: RngMPolElt, RngMPolElt -> RngMPolElt`

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

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

`a / b: RngMPolElt, RngMPolElt -> FldFunMElt`

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

`a +:= b: RngMPolElt, RngMPolElt -> RngMPolElt`

`a -:= b: RngMPolElt, RngMPolElt -> RngMPolElt`

`a *:= b: RngMPolElt, RngMPolElt -> RngMPolElt`

`a div:= b: RngMPolElt, RngMPolElt -> RngMPolElt`

25.4.2 Equality and Membership

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

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

`a in R: RngMPolElt, Rng -> BoolElt`

`a notin R: RngMPolElt, Rng -> BoolElt`

25.4.3 Predicates on Ring Elements

The list belows contains the general ring element predicates. Also, the `IsDivisibleBy` function allows a divisibility test, and the `IsAlgebraicallyDependent` function determines if a set of ring elements is algebraically dependent. Note that not all functions are available for every coefficient ring.

`IsDivisibleBy(a, b): RngMPolElt, RngMPolElt -> BoolElt, RngMPolElt`

Given elements a, b in a multivariate polynomial ring P, this function returns whether the polynomial a is divisible by b in P, that is, if and only if there exists $q \in P$ such that $a = q \cdot b$. If `true` is returned, the quotient polynomial q is also returned.

`IsAlgebraicallyDependent(S): {RngMPolElt} -> BoolElt`

Returns `true` iff the set S of multivariate polynomials is algebraically dependent.

`IsZero(f): RngMPolElt -> BoolElt`

`IsOne(f): RngMPolElt -> BoolElt`

`IsMinusOne(f): RngMPolElt -> BoolElt`

`IsNilpotent(f): RngMPolElt -> BoolElt`

`IsIdempotent(f): RngMPolElt -> BoolElt`

`IsUnit(f): RngMPolElt -> BoolElt`

`IsZeroDivisor(f): RngMPolElt -> BoolElt`

`IsRegular(f): RngMPolElt -> BoolElt`

`IsIrreducible(f): RngMPolElt -> BoolElt`

`IsPrime(f): RngMPolElt -> BoolElt`

25.4.4 Coefficients, Monomials, and Terms

Many of the functions in this subsection come in three different forms: one in which no variable is specified, which usually returns values in the coefficient ring, and two in which a particular variable is referred, either by name or by number, and these usually return values in the polynomial ring itself.

`Coefficients(f): RngMPolElt -> [ RngElt ]`

Given a multivariate polynomial f with coefficients in R, this function returns a sequence of 'base' coefficients, that is, a sequence of elements of R occurring as coefficients of the monomials in f. Note that the monomials are ordered, and that the sequence of coefficients corresponds exactly to the sequence of monomials returned by `Monomials(f)`.

`Coefficients(f, i): RngMPolElt, RngIntElt -> [ RngElt ]`

`Coefficients(f, v): RngMPolElt, RngMPolElt -> [ RngElt ]`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns a sequence of coefficients with respect to a given variable $v = x_i$, that is, the function returns a sequence of elements of P that form the coefficients of the powers of v (in ascending order) when f is regarded as a polynomial $\sum_j c_j x_i^j$; note that the variable x_i itself will not occur in the coefficients. There are two ways to indicate with respect to which variable the coefficients are to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Coefficient(f, i, k): RngMPolElt, RngIntElt, RngIntElt -> RngElt`

`Coefficient(f, v, k): RngMPolElt, RngMPolElt, RngIntElt -> RngElt`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the coefficient of $v^k = x_i^k$, that is, the function returns the element of P that forms the coefficient of the k-th power of x_i, when f is regarded as a polynomial $\sum_j c_j x_i^j$; note that the variable x_i itself will not occur in the coefficient. There are two ways to indicate with respect to which variable the coefficient is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`LeadingCoefficient(f): RngMPolElt -> RngElt`

Given a multivariate polynomial f with coefficients in R, this function returns the leading coefficient of f as an element of R; this is the coefficient of the leading monomial of f, that is, the first among the monomials occurring in f with respect to the ordering of monomials used in P.

`LeadingCoefficient(f, i): RngMPolElt, RngIntElt -> RngElt`

`LeadingCoefficient(f, v): RngMPolElt, RngMPolElt -> RngElt`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the element of P that forms the coefficient of the largest power of $v = x_i$ that occurs with non-zero coefficient in f, when f is regarded as a polynomial $\sum_j c_j x_i^j$; note that the variable x_i itself will not occur in the coefficient. There are two ways to indicate with respect to which variable the leading coefficient is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Length(f): RngMPolElt -> RngIntElt`

Given a multivariate polynomial f, return the length of f, i.e., the number of terms of f.

`TrailingCoefficient(f): RngMPolElt -> RngElt`

Given a multivariate polynomial f with coefficients in R, this function returns the trailing coefficient of f as an element of R; this is the coefficient of the trailing monomial of f, that is, the last among the monomials occurring in f with respect to the ordering of monomials used in P.

`TrailingCoefficient(f, i): RngMPolElt, RngIntElt -> RngElt`

`TrailingCoefficient(f, v): RngMPolElt, RngMPolElt -> RngElt`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the element of P that forms the coefficient of the least power of $v = x_i$ that occurs with non-zero coefficient in f, when f is regarded as a polynomial $\sum_j c_j x_i^j$; note that the variable x_i itself will not occur in the coefficient. There are two ways to indicate with respect to which variable the leading coefficient is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`CoefficientDenominator(f): RngMPolElt -> RngElt`

`CoefficientNumerator(f): RngMPolElt -> RngElt`

Given a multivariate polynomial f with coefficients in $\mathbb{Q}$, a rational function field, or a field of fractions of an order of a number or function field, return the LCM of the denominators of the coefficients of f (`CoefficientDenominator`) or the product of that LCM with f itself (`CoefficientNumerator`).

`MonomialCoefficient(f, m): RngMPolElt, RngMPolElt -> RngElt`

Given a multivariate polynomial f and a monomial m, both in $P \in R[x_1, \ldots, x_n]$, this function returns the coefficient with which m occurs in f as an element of R.

`Monomials(f): RngMPolElt -> [ RngMPolElt ]`

Given a multivariate polynomial $f \in P$, this function returns a sequence of monomials, that is, a sequence of monomial elements of P occurring in f. Note that the monomials in P are ordered, and that the sequence of monomials corresponds exactly to the sequence of coefficients returned by `Coefficients(f)`.

`CoefficientsAndMonomials(f): RngMPolElt -> [ RngElt ], [ RngMPolElt ]`

Given a multivariate polynomial $f \in P$, this function returns parallel sequences C and M of the coefficients and monomials, respectively, of f. Thus this function is equivalent to calling

`Coefficients` and `Monomials` separately, but is more efficient (particularly for large polynomials) since only one scan of the polynomial needs to be done.

`LeadingMonomial(f): RngMPolElt -> RngMPolElt`

Given a multivariate polynomial $f \in P$ this function returns the leading monomial of f, that is, the first monomial element of P that occurs in f, with respect to the ordering of monomials used in P.

`Terms(f): RngMPolElt -> [ RngMPolElt ]`

Given a multivariate polynomial $f \in P$, this function returns the sequence of (non-zero) terms of f as elements of P. The terms are ordered according to the ordering on the monomials in P. Consequently the i-th element of this sequence of terms will be equal to the product of the i-th element of the sequence of coefficients and the i-th element of the sequence of monomials.

`Terms(f, i): RngMPolElt, RngIntElt -> [ RngMPolElt ]`

`Terms(f, v): RngMPolElt, RngMPolElt -> [ RngMPolElt ]`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns a sequence of terms with respect to a given variable $v = x_i$, that is, the function returns a sequence of elements of P that form the terms (ascending order) of f regarded as a polynomial $\sum_j c_j x_i^j$. There are two ways to indicate with respect to which variable the terms are to be ordered: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Term(f, i, k): RngMPolElt, RngIntElt, RngIntElt -> RngMPolElt`

`Term(f, v, k): RngMPolElt, RngMPolElt, RngIntElt -> RngMPolElt`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the k-th term of f (with $k \geq 0$), that is, the function returns the term of f involving the k-th power of x_i, when f is regarded as a polynomial $\sum_j c_j x_i^j$. There are two ways to indicate with respect to which variable the term is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`LeadingTerm(f): RngMPolElt -> RngMPolElt`

Given a multivariate polynomial $f \in P$, this function returns the leading term of f as an element of P; this is the product of the leading monomial and the leading coefficient that is, the first among the monomial terms occurring in f with respect to the ordering of monomials used in P.

`LeadingTerm(f, i): RngMPolElt, RngIntElt -> RngMPolElt`

`LeadingTerm(f, v): RngMPolElt, RngMPolElt -> RngMPolElt`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the element of P that forms the leading term of f when f is regarded as a polynomial $\sum_j c_j x_i^j$. Thus it is the term involving the largest power of x_i that occurs with non-zero coefficient. There are two ways to indicate with respect to which variable the leading coefficient is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`TrailingTerm(f): RngMPolElt -> RngElt`

Given a multivariate polynomial $f \in P$, this function returns the trailing term of f as an element of P; this is the last among the monomial terms occurring in f with respect to the ordering of monomials used in P.

`TrailingTerm(f, i): RngMPolElt, RngIntElt -> RngElt`

`TrailingTerm(f, v): RngMPolElt, RngMPolElt -> RngElt`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the element of P that forms the trailing term of f when f is regarded as a polynomial $\sum_j c_j x_i^j$. Thus it is the term involving the least power of x_i that occurs with non-zero coefficient. There are two ways to indicate with respect to which variable the leading coefficient is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Exponents(f): RngMPolElt -> [ RngIntElt ]`

Given a single term f (a polynomial having exactly one term) in a polynomial ring of rank n, return the exponents of the monomial of f, as a sequence of length n of integers. (The coefficient of f is ignored; it need not be 1.)

`Monomial(P, E): RngMPol, [ RngIntElt ] -> RngMPolElt`

Given a multivariate polynomial ring $P = R[x_1, \ldots, x_n]$, and a sequence E of non-negative integers, return the monomial ${x_1}^{E[1]} \ldots {x_n}^{E[n]}$ in P. This function is a semi-inverse of `Exponents`.

`Polynomial(C, M): [RngElt], [RngMPolElt] -> RngMPolElt`

Given a length-k sequence C of coefficients in a ring R and a length-k sequence M of monomials of a polynomial ring R, return the multivariate polynomial $f \in R$ whose coefficients are C and monomials are M. (Thus for any $f \in R$, `Polynomial(Coefficients(f), Monomials(f))` equals f.)

Example: Coefficients and Monomials (ex-f062e9)

In this and the next example we illustrate the coefficient and term functions, using the polynomial in three variables x, y, z over the rational field that is given by $f = (2x + y)z^3 + 11xyz + x^2y^2$.

```
> R<x, y, z> := PolynomialAlgebra(RationalField(), 3);
> f := (2*x+y)*z^3+11*x*y*z+x^2*y^2;
> f;
x^2*y^2 + 11*x*y*z + 2*x*z^3 + y*z^3
> Coefficients(f);
[ 1, 11, 2, 1 ]
> Monomials(f);
[
   x^2*y^2,
   x*y*z,
   x*z^3,
   y*z^3
]
> CoefficientsAndMonomials(f);
[ 1, 11, 2, 1 ]
[
    x^2*y^2,
    x*y*z,
    x*z^3,
    y*z^3
]
> Terms(f);
[
   x^2*y^2,
   11*x*y*z,
   2*x*z^3,
   y*z^3
]
> Coefficients(f, y);
[
   2*x*z^3,
   11*x*z + z^3,
   x^2
]
> Terms(f, 2);
[
   2*x*z^3,
   11*x*y*z + y*z^3,
   x^2*y^2
```

(continues on next page)

(continued from previous page)

```
]
> MonomialCoefficient(f, x*y*z);
11
> LeadingTerm(f);
x^2*y^2
> LeadingTerm(f, z);
2*x*z^3 + y*z^3
> LeadingCoefficient(f, z);
2*x + y
> Polynomial([1, 2, 3], [x*y, y, z^2]);
x*y + 2*y + 3*z^2
```

25.4.5 Degrees

`Degree(f, i): RngMPolElt, RngIntElt -> RngIntElt`

`Degree(f, v): RngMPolElt, RngMPolElt -> RngIntElt`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the degree of f in $v^k = x_i^k$, that is, the function returns the degree of f when it is regarded as a polynomial $\sum_j c_j x_i^j$. The resulting integer is thus the largest power of x_i occurring in any monomial of f. There are two ways to indicate with respect to which variable the degree is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P). If f is the zero polynomial, the return value is always -1.

`TotalDegree(f): RngMPolElt -> RngIntElt`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the total degree of f, which is the maximum of the total degrees of all monomials that occur in f. The total degree of a monomial m is the sum of the exponents of the indeterminates that make up m. Note that this ignores the weights on the variables if there are any (see the section on graded polynomial rings below). If f is the zero polynomial, the return value is -1.

`LeadingTotalDegree(f): RngMPolElt -> RngIntElt`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the leading total degree of f, which is the total degree of the leading monomial of f. If f is the zero polynomial, the return value is -1.

25.4.6 Univariate Polynomials

`IsUnivariate(f): RngMPolElt -> BoolElt, RngUPolElt, RngIntElt`

Given a multivariate polynomial $f \in R[x_1, \ldots, x_n]$, this function returns whether f is in fact a univariate polynomial in one of its indeterminates $x_1, \ldots, x_n$. If true is returned, then the function also returns a univariate version u of f and (the first) i such that f is univariate in x_i. Note that there will only be ambiguity about i if f is a constant polynomial. The univariate polynomial u will be an element of $R[x]$ with the same coefficients as f.

`IsUnivariate(f, i): RngMPolElt, RngIntElt -> BoolElt, RngUPolElt`

`IsUnivariate(f, v): RngMPolElt, RngMPolElt -> BoolElt, RngUPolElt`

Given a multivariate polynomial $f \in R[x_1, \ldots, x_n]$, this function returns whether f is in fact a univariate polynomial in x_i. If true is returned, then the function also returns a univariate version u of f, which will be an element of the univariate polynomial ring $R[x]$ with the same coefficients as f. The indeterminate x_i should either be specified as a (polynomial) argument v or as an integer i.

`UnivariatePolynomial(f): RngMPolElt -> RngUPolElt`

Given a multivariate polynomial $f \in R[x_1, \ldots, x_n]$, which is known to be a univariate polynomial in x_i for some i with $1 \leq i \leq n$, return a univariate version u of f, which will be an element of the univariate polynomial ring $R[x]$ with the same coefficients as f.

Example: Converting to a Univariate Polynomial (ex-f54865)

Suppose we have two bivariate polynomials f and g over some ring.

```
> P<x,y> := PolynomialRing(GF(5), 2);
> f := x^2 - y + 3;
> g := y^3 - x*y + x;
```

If we compute the resultant in either variable of the two polynomials, then we can apply `UnivariatePolynomial` to this to obtain a univariate version of it, from which we can compute the roots.

```
> ry := Resultant(f, g, y);
> ry;
4*x^6 + x^4 + x^3 + 3*x^2 + 2*x + 3
> Roots(UnivariatePolynomial(ry));
[ <3, 1> ]
> Evaluate(f, x, 3);
4*y + 2
> Evaluate(g, x, 3);
y^3 + 2*y + 3
> GCD($1, $2);
y + 3
> rx := Resultant(f, g, x);
> rx;
y^6 + 4*y^3 + 3*y + 3
> Roots(UnivariatePolynomial(rx));
[ <2, 1> ]
> Evaluate(f, y, 2);
x^2 + 1
> Evaluate(g, y, 2);
4*x + 3
> GCD($1, $2);
x + 2
```

25.4.7 Derivative, Integral

`Derivative(f, i): RngMPolElt, RngIntElt -> RngMPolElt`

`Derivative(f, v): RngMPolElt, RngMPolElt -> RngMPolElt`

Given a multivariate polynomial $f \in P$, return the derivative of f with respect to the variable $v = x_i$, as an element of P. There are two ways to indicate with respect to which variable the derivative is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Derivative(f, k, i): RngMPolElt, RngIntElt -> RngMPolElt`

`Derivative(f, k, v): RngMPolElt, RngMPolElt -> RngMPolElt`

Given a multivariate polynomial $f \in P$ and an integer $k > 0$, return the k-th derivative of f with respect to the variable $v = x_i$, as an element of P. There are two ways to indicate with respect to which variable the derivative is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Integral(f, i): RngMPolElt, RngIntElt -> RngMPolElt`

`Integral(f, v): RngMPolElt, RngMPolElt -> RngMPolElt`

Given a multivariate polynomial $f \in P$ over a field of characteristic zero, return the formal integral of f with respect to $v = x_i$ as an element of P. There are two ways to indicate with respect to which variable the integral is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the

number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`JacobianMatrix([ f ]): [ RngMPolElt ] -> RngMPol`

Creates the matrix with (i, j)'th entry the partial derivative of the i'th polynomial in the list with the j'th indeterminate of its parent ring.

25.4.8 Evaluation, Interpolation

`Evaluate(f, s): RngMPolElt, [ RngElt ] -> RngElt`

`Evaluate(f, s): RngMPolElt, < RngElt, ..., RngElt > -> RngElt`

Given an element f of a polynomial ring $P = R[x_1, \ldots, x_n]$ and a sequence or tuple s of ring elements of length n, return the value of f at s, that is, obtained by substituting $x_i = s[i]$. If the elements of s can be lifted into the coefficient ring R, then the result will be an element of R. If the elements of s cannot be lifted to the coefficient ring, then an attempt is made to do a generic evaluation of f at s. In this case, the result will be of the same type as the elements of s.

`Evaluate(f, i, r): RngMPolElt, RngMPolElt, RngElt -> RngMPolElt`

`Evaluate(f, v, r): RngMPolElt, RngMPolElt, RngElt -> RngMPolElt`

Given an element f of a multivariate polynomial ring P and a ring element r return the value of f when the variable $v = x_i$ is evaluated at r. If r can be coerced into the coefficient ring of P, the result will be an element in P again. Otherwise the other variables of P must be coercible into the parent of r, and the result will have the same parent as r.

`Interpolation(I, V, i): [ RngElt ], [ RngMPolElt ], RngIntElt -> RngMPolElt`

`Interpolation(I, V, v): [ RngElt ], [ RngMPolElt ], RngIntElt -> RngMPolElt`

Let K be a field, and $P = K[x_1, \ldots, x_n]$ a multivariate polynomial ring over K; let $v = x_i$ be the i-th indeterminate of P. Given a sequence I of elements of K (the interpolation points) and a sequence V of elements of P (the interpolation values), both sequences of length $k > 0$, return the unique polynomial $f \in P$ of degree less than k in the variable x_i such that $f(I[j]) = V[j]$, for $j = 1, \ldots, k$. The variable x_i may not occur anywhere in the values V. There are two ways to indicate with respect to which variable to interpolate: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable or the variable v itself (as a polynomial).

Example: Polynomial Interpolation (ex-ca9d39)

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

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

25.4.9 Quotient and Reductum

`f div g: RngMPolElt, RngMPolElt -> RngMPolElt`

`ExactQuotient(f, g): RngMPolElt, RngMPolElt -> RngMPolElt`

The quotient of the multivariate polynomial f by g in $R[x_1, \ldots, x_n]$, provided the result lies in P again. If a polynomial q in P exists such that $f = q \cdot g$ then it will be returned, but if none exists an error results. When R is a domain the quotient is unique and is always found. Over a ring with

zero divisors the quotient need not be unique, and the function may report that g does not divide f even when it does.

Reductum(f): RngMPolElt -> RngMPolElt

The reductum of a polynomial f, which is the polynomial obtained by removing the leading term of f.

Reductum(f, i): RngMPolElt, RngIntElt -> RngMPolElt

Reductum(f, v): RngMPolElt, RngMPolElt -> RngMPolElt

The reductum of a multivariate polynomial $f \in R[x_1, \ldots, x_n]$ obtained by removing the leading term with respect to the variable $v = x_i$. Here either v must be specified as a polynomial, or x_i must be specified by providing the integer i, with $1 \leq i \leq n$.

25.4.10 Diagonalizing a Polynomial of Degree 2

We provide two basic tools that deal with polynomial diagonalization.

SymmetricBilinearForm(f): RngMPolElt -> ModMatRngElt

The symmetric bilinear form (as a matrix) of a multivariate polynomial of degree 2.

DiagonalForm(f): RngMPolElt -> RngMPolElt, ModMatRngElt

The diagonal form of the multivariate polynomial of degree 2. Also returns the transformation matrix.

Example: Symmetric Bilinear Form of a Quadratic Polynomial (ex-649fdb)

```
> Q := RationalField();
> PR<x, y, z> := PolynomialRing(Q, 3);
> g := 119/44*x^2 - 93759/41440*x*y + 390935/91427*x*z
>      + 212/243*x - 3/17*y^2 + 52808/172227*y*z
>      - 287/227*y + 537/934*z^2 - 127/422*z;
> SymmetricBilinearForm(g);
[       119/44  -93759/82880 390935/182854       106/243]
[ -93759/82880         -3/17  26404/172227      -287/454]
[390935/182854  26404/172227       537/934      -127/844]
[      106/243      -287/454      -127/844             0]
> DiagonalForm(g);
119/44*x^2 - 15798558582429/4*y^2 +
34932799628335074761085292707227419544217/934*z^2 -
17658873286101893452437121055688364561927521739811614723483771045740414637 1/2
>
> bl := SymmetricBilinearForm(g);
> NBL := Matrix(PR, bl);
> D, T := OrthogonalizeGram(bl);
> NT := Matrix(PR, T);
> C := Matrix(PR, [[x,y,z,1]]);
> NC := C * NT;
> NCT := Transpose(NC);
> (NC * NBL * NCT)[1][1] eq DiagonalForm(g);
true
```

The last few statements demonstrate how the polynomial's diagonal form is obtained from its symmetric bilinear form. Note also that since the polynomial g is not homogeneous its symmetric bilinear form is given on four variables, the fourth variable being a homogenizing variable.

25.5 Greatest Common Divisors

The functions in this section can be applied to multivariate polynomials over any ring which has a GCD algorithm.

25.5.1 Common Divisors and Common Multiples

GreatestCommonDivisor(f, g): RngMPolElt, RngMPolElt -> RngMPolElt

Gcd(f, g): RngMPolElt, RngMPolElt -> RngMPolElt

GCD(f, g): RngMPolElt, RngMPolElt -> RngMPolElt

The greatest common divisor of f and g in a multivariate polynomial ring P. If either of the inputs is zero, then the result is the other input (and if the inputs are both zero then the result is zero). The result is normalized (see the function *Normalize*), so the result is always unique.

The valid coefficient rings are those which themselves have a GCD algorithm for their elements (which includes most commutative rings in Magma).

For polynomials over the integers or rationals, a combination of three algorithms is used: (1) the heuristic evaluation 'GCDHEU' algorithm of Char et al. ([CGG89] and [GCL92, section 7.7]), suitable for moderate-degree dense polynomials with several variables; (2) the EEZ-GCD algorithm of Wang ([MY73, Wan80] and [GCL92, section 7.6]), based on evaluation and sparse ideal-adic multivariate Hensel lifting ([Wan78] and [GCL92, section 6.8]), suitable for sparse polynomials; (3) a recursive multivariate evaluation-interpolation algorithm (similar to that in [GCL92, section 7.4]), which in fact works generically over $\mathbb{Z}$ or most fields.

For polynomials over any finite field or any field of characteristic zero besides $\mathbb{Q}$, the generic recursive multivariate evaluation-interpolation algorithm (3) above is used, which effectively takes advantage of any fast modular algorithm for the base univariate polynomials (e.g., for number fields). See the function *GreatestCommonDivisor* in the univariate polynomials chapter for details of univariate GCD algorithms.

For polynomials over another polynomial ring or rational function field, the polynomials are first "flattened" to be inside a multivariate polynomial ring over the base coefficient ring, then the appropriate algorithm is used for that base coefficient ring.

For polynomials over any other ring, the generic subresultant algorithm [Coh93, section 3.3] is called recursively on a subring with one less variable.

GCD(Q): [RngMPolElt] -> RngMPolElt

Given a sequence Q of polynomials, return the GCD of the elements of Q. If Q has length 0 and universe P, then the zero element of P is returned.

LeastCommonMultiple(f, g): RngMPolElt, RngMPolElt -> RngMPolElt

Lcm(f, g): RngMPolElt, RngMPolElt -> RngMPolElt

LCM(f, g): RngMPolElt, RngMPolElt -> RngMPolElt

The least common multiple of f and g in a multivariate polynomial ring P. The LCM of zero and anything else is zero. The result is normalized (see the function *Normalize*), so the result is always unique. The valid coefficient rings are as for the function GCD, above.

The LCM is effectively computed as `Normalize((F div GCD(F, G)) * G)`, for non-zero inputs.

LCM(Q): [RngMPolElt] -> RngMPolElt

Given a sequence Q of polynomials, return the LCM of the elements of Q. If Q has length 0 and universe P, then the one element of P is returned.

Normalize(f): RngMPolElt -> RngMPolElt

Given a polynomial f over the base ring R, this function returns the unique normalized polynomial g which is associate to f (so $g = uf$ for some unit in R). This is chosen so that if R is a field then g is monic, if R is $\mathbb{Z}$ then the leading coefficient of g is positive, if R is a polynomial ring itself, then the leading coefficient of g is recursively normalized, and so on for other rings.

`ClearDenominators(f): RngMPolElt -> RngMPolElt`

`ClearDenominators(Q): [ RngMPolElt ] -> [ RngMPolElt ]`

Given a polynomial f over a field K such that K is the field of fractions of a domain D, the first function computes the lowest common denominator L of the coefficients of f and returns the polynomial $g = L \cdot f$ over D with cleared denominators, and L. The second function returns the sequence of polynomials derived from independently clearing the denominators in each polynomial in the given sequence Q.

25.5.2 Content and Primitive Part

`Content(f): RngMPolElt -> RngIntElt`

The content of f, that is, the greatest common divisor of the coefficients of f as an element of the coefficient ring.

`PrimitivePart(f): RngMPolElt -> RngMPolElt`

The primitive part of f, being f divided by the content of f.

`ContentAndPrimitivePart(f): RngMPolElt -> RngIntElt, RngMPolElt`

`Contpp(f): RngMPolElt -> RngIntElt, RngMPolElt`

The content (the greatest common divisor of the coefficients) of f, as an element of the coefficient ring, as well as the primitive part (f divided by the content) of f.

25.6 Factorization and Irreducibility

We describe the functions for multivariate polynomial factorization and associated computations.

`Factorization(f): RngMPolElt -> [ < RngMPolElt, RngIntElt >], RngElt`

Given a multivariate polynomial f over the ring R, this function returns the factorization of f as a factorization sequence Q, that is, a sequence of pairs, each consisting of an irreducible factor q_i a positive integer k_i (its multiplicity). Each irreducible factor is normalized (see the function *Normalize*), so the expansion of the factorization sequence is the unique canonical associate of f. The function also returns the unit u of R giving the normalization, so $f = u \cdot \prod_i q_i^{k_i}$.

The coefficient ring R must be one of the following: a finite field $\mathbb{F}_q$, the ring of integers $\mathbb{Z}$, the field of rationals $\mathbb{Q}$, an algebraic number field $\mathbb{Q}(\alpha)$, or a polynomial ring, function field (rational or algebraic) or finite-dimensional affine algebra (which is a field) over any of the above.

For bivariate polynomials, a polynomial-time algorithm in the same spirit as van Hoeij's Knapsack factoring algorithm [vH02] is used.

For polynomials over the integers or rationals, an algorithm similar to that presented in [Wan78] and [GCL92, section 6.8], based on evaluation and sparse ideal-adic multivariate Hensel lifting, is used.

For polynomials over any finite field, a similar algorithm is used, with a few special modifications for non-zero characteristic (see, for example, [BM97]).

For polynomials over algebraic number fields and affine algebras, a multivariate version of the norm-based algorithm of Trager [Tra76] is used, which performs a suitable substitution and multivariate resultant computation, and then factors the resulting integral multivariate polynomial.

Each of these algorithms reduces to univariate factorization over the base ring; for details of how this factorization is done in each case, see the function *Factorization* in the univariate polynomial rings chapter.

For polynomials over another polynomial ring or function field, the polynomials are first "flattened" to be inside a multivariate polynomial ring over the base coefficient ring, then the appropriate algorithm is used for that base coefficient ring.

`SquarefreeFactorization(f): RngMPolElt -> [ <RngMPolElt, RngIntElt> ]`

Return the squarefree factorization of the multivariate polynomial f as a sequence of tuples of length 2, each consisting of a (not necessarily irreducible) factor and an integer indicating the multiplicity. The factors do not contain the square of any polynomial of degree greater than 0. The allowable coefficient rings are the same as those allowable for the function `Factorization`.

`SquarefreePart(f): RngMPolElt -> RngMPolElt`

Return the squarefree part of the multivariate polynomial f, which is the largest (normalized) divisor g of f which is squarefree.

`IsIrreducible(f): RngMPolElt -> BoolElt`

Given a multivariate polynomial f over a ring R, this function returns whether f is irreducible over R. The allowable coefficient rings are the same as those allowable for the function `Factorization`.

`SetVerbose("PolyFact", v): MonStgElt, RngIntElt`

(Procedure.) Change the verbose printing level for all polynomial factorization algorithms to be v. Currently the legal levels are 0, 1, 2, or 3.

Example: Factorizing a Product of Trinomials (ex-2f44bb)

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

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> f := &*[x^i+y^j+z^k: i,j,k in [1..2]];
> #Terms(f);
461
> TotalDegree(f);
15
> time Factorization(f);
[
    <x + y + z, 1>,
    <x + y + z^2, 1>,
    <x + y^2 + z, 1>,
    <x + y^2 + z^2, 1>,
    <x^2 + y + z, 1>,
    <x^2 + y + z^2, 1>,
    <x^2 + y^2 + z, 1>,
    <x^2 + y^2 + z^2, 1>
]
Time: 0.290
```

Example: Factorizing a Vandermonde Determinant (ex-12897c)

We construct a Vandermonde matrix of rank 6, find its determinant, and factorize that determinant.

```
> // Create polynomial ring over R of rank n
> PRing := function(R, n)
>     P := PolynomialRing(R, n);
>     AssignNames(~P, ["x" cat IntegerToString(i): i in [1..n]]);
>     return P;
> end function;
>
> // Create Vandermonde matrix of rank n
> Vandermonde := function(n)
>     P := PRing(IntegerRing(), n);
>     return MatrixRing(P, n) ! [P.i^(j - 1): i, j in [1 .. n]];
> end function;
```

(continues on next page)

(continued from previous page)

```
>
> V := Vandermonde(6);
> V;
[   1   x1 x1^2 x1^3 x1^4 x1^5]
[   1   x2 x2^2 x2^3 x2^4 x2^5]
[   1   x3 x3^2 x3^3 x3^4 x3^5]
[   1   x4 x4^2 x4^3 x4^4 x4^5]
[   1   x5 x5^2 x5^3 x5^4 x5^5]
[   1   x6 x6^2 x6^3 x6^4 x6^5]
> D := Determinant(V);
> #Terms(D);
720
> TotalDegree(D);
15
> time Factorization(D);
[
    <x5 - x6, 1>,
    <x4 - x6, 1>,
    <x4 - x5, 1>,
    <x3 - x6, 1>,
    <x3 - x5, 1>,
    <x3 - x4, 1>,
    <x2 - x6, 1>,
    <x2 - x5, 1>,
    <x2 - x4, 1>,
    <x2 - x3, 1>,
    <x1 - x6, 1>,
    <x1 - x5, 1>,
    <x1 - x4, 1>,
    <x1 - x3, 1>,
    <x1 - x2, 1>
]
Time: 0.030
```

Example: Heron's Formula for the Area of a Triangle (ex-f94225)

We construct a polynomial $A2$ in three indeterminates a, b, and c over the rational field such that $A2$ is the square of the area of the triangle with side lengths a, b, c. Using elementary trigonometry one can derive the expression $(4 * a^2 * b^2 - (a^2 + b^2 - c^2)^2)/16$ for $A2$. Factorizing $A2$ gives a nice formulation of the square of the area which is similar to that given by *Heron's formula*.

```
> P<a, b, c> := PolynomialRing(RationalField(), 3);
> A2 := 1/16 * (4*a^2*b^2 - (a^2 + b^2 - c^2)^2);
> A2;
-1/16*a^4 + 1/8*a^2*b^2 + 1/8*a^2*c^2 - 1/16*b^4 + 1/8*b^2*c^2 - 1/16*c^4
> F, u := Factorization(A2);
> F;
[
    <a - b - c, 1>,
    <a - b + c, 1>,
    <a + b - c, 1>,
    <a + b + c, 1>
]
> u;
-1/16
```

Example: Finite Field Factorization (ex-8e8a69)

We factorize a multivariate polynomial over a finite field.

```
> Frob := function(G)
>     n := #G;
>     I := {@ g: g in G @};
>     P := PolynomialRing(GF(2), n);
>     AssignNames(~P, [CodeToString(96 + i): i in [1 .. n]]);
>     M := MatrixRing(P, n);
>     return M ! &cat[
>         [P.Index(I, I[i] * I[j]): j in [1 .. n]]: i in [1 .. n]
>     ];
> end function;
> A := Frob(Sym(3));
> A;
[a b c d e f]
[b c a f d e]
[c a b e f d]
[d e f a b c]
[e f d c a b]
[f d e b c a]
> Determinant(A);
a^6 + a^4*d^2 + a^4*e^2 + a^4*f^2 + a^2*b^2*c^2 +
    a^2*b^2*d^2 + a^2*b^2*e^2 + a^2*b^2*f^2 + a^2*c^2*d^2 +
    a^2*c^2*e^2 + a^2*c^2*f^2 + a^2*d^4 + a^2*d^2*e^2 +
    a^2*d^2*f^2 + a^2*e^4 + a^2*e^2*f^2 + a^2*f^4 + b^6 +
    b^4*d^2 + b^4*e^2 + b^4*f^2 + b^2*c^2*d^2 + b^2*c^2*e^2
    + b^2*c^2*f^2 + b^2*d^4 + b^2*d^2*e^2 + b^2*d^2*f^2 +
    b^2*e^4 + b^2*e^2*f^2 + b^2*f^4 + c^6 + c^4*d^2 +
    c^4*e^2 + c^4*f^2 + c^2*d^4 + c^2*d^2*e^2 + c^2*d^2*f^2
    + c^2*e^4 + c^2*e^2*f^2 + c^2*f^4 + d^6 + d^2*e^2*f^2 +
    e^6 + f^6
> time Factorization(Determinant(A));
[
    <a + b + c + d + e + f, 2>,
    <a^2 + a*b + a*c + b^2 + b*c + c^2 + d^2 + d*e + d*f +
        e^2 + e*f + f^2, 2>
]
Time: 0.049
```

25.7 Resultants and Discriminants

Resultant(f, g, i): RngMPolElt, RngMPolElt, RngIntElt -> RngMPolElt

Resultant(f, g, v): RngMPolElt, RngMPolElt, RngMPolElt -> RngMPolElt

The resultant of multivariate polynomials f and g in $P = R[x_1, \ldots, x_n]$ with respect to the variable $v = x_i$, which is by definition the determinant of the Sylvester matrix for f and g when considered as polynomials in the single variable x_i. The result will be an element of P again. There are two ways to indicate with respect to which variable the integral is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P). The algorithm used is the modular interpolation method, as given in [GCL92, pp. 412--413].

Discriminant(f, i): RngMPolElt, RngIntElt -> RngMPolElt

Discriminant(f, v): RngMPolElt, RngMPolElt -> RngMPolElt

The discriminant D of $f \in R[x_1, \ldots, x_n]$ is returned, where f is considered as a polynomial in $v = x_i$. The result will be an element of P again. The coefficient ring R must be a domain, or the leading coefficient of f in v must be a unit. There are two ways to indicate with respect to which variable the integral is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number

of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

25.8 Polynomials over the Integers

The functions in this section are available for multivariate polynomials over the integers only.

`Sign(f): RngMPolElt -> RngIntElt`

The sign of the leading coefficient of f.

`AbsoluteValue(f): RngMPolElt -> RngMPolElt`

`Abs(f): RngMPolElt -> RngMPolElt`

Return either f or $-f$ according to which one has non-negative leading coefficient.

`MaxNorm(f): RngMPolElt -> RngIntElt`

The maximum of the absolute values of the coefficients of f.

`SumNorm(f): RngMPolElt -> RngIntElt`

The sum of the base coefficients of f.

25.9 Bibliography

- [BM97] Laurent Bernardin and Michael B. Monagan. Efficient Multivariate Factorization Over Finite Fields. In *Proceedings of AAECC*, volume 1255 of LNCS, 15–28. Springer-Verlag, 1997.
- [CGG89] Bruce W. Char, Keith O. Geddes, and Gaston H. Gonnet. GCDHEU: Heuristic Polynomial GCD Algorithm Based on Integer GCD Computation. *J. Symbolic Comp.*, 7(1):31–48, 1989.
- [Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory.* Volume 138 of Graduate Texts in Mathematics. Springer, Berlin–Heidelberg–New York, 1993.
- [GCL92] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra.* Kluwer, Boston/Dordrecht/London, 1992.
- [MY73] J. Moses and D.Y.Y. Yun. The EZ GCD algorithm. *Proc. ACM Annual Conference*, 73(2):159–166, 1973.
- [Tra76] Barry M. Trager. Algebraic factoring and rational function integration. In R.D. Jenks, editor, *Proc. SYMSAC '76*, 196–208. ACM press, 1976.
- [vH02] Mark van Hoeij. Factoring Polynomials and the knapsack problem. *J. Number Th.*, 95(2):167–189, 2002. http://www.math.fsu.edu/∼hoeij/paper/knapsack.ps.
- [Wan78] Paul S. Wang. An improved multivariate polynomial factoring algorithm. *Math. Comp.*, 32(144):1215–1231, 1978.
- [Wan80] Paul S. Wang. The EEZ-GCD algorithm. *SIGSAM Bulletin*, 14(2):50–60, 1980.

Chapter 26

Real and Complex Fields

26.1 Introduction

Real and complex numbers can only be stored in the computer effectively in approximations. Magma provides a number of facilities for calculating with such approximations. Most of these facilities are based upon the C libraries MPFR, which provides algorithms for manipulating real numbers with *exact rounding*, and MPC, an extension of MPFR to handle complex numbers. More specifically, the MPFR library extends the semantics of the ANSI/IEEE-754 standard for double-precision numbers — which are used in virtually all major programming languages — to handle real numbers of arbitrary precision. The precise semantics of MPFR give the user fine control over precision loss, which is a tremendous advantage when working with reals and complexes. Magma currently uses MPFR 2.4.1 and MPC 0.8. Documentation for algorithms used in MPFR can be found at `mpfr.org`.

As MPFR and MPC are works in progress, they do not yet provide a complete framework for working with the reals and complexes. For those functions that these libraries are missing, Magma falls back to algorithms taken from Pari. The documentation of MPFR and MPC provide a list of the functions that they provide. Assume that each intrinsic uses MPFR unless otherwise stated.

Although we use the terms *real field* and *complex field* for Magma structures containing real or complex approximations, it should be noted that such a subset of the real or complex field may not even form a commutative ring. Never the less, the real and complex fields are considered to be fields by Magma, they comprise objects of type `FldRe` and `FldCom` with elements of type `FldReElt` and `FldComElt` respectively.

26.1.1 Real Numbers in Magma

Real numbers are stored internally as a sign bit and an expansion $\sum b_i 2^i$ with $b_i \in 0, 1$. Note that MPFR has both positive and negative zeros. Complex numbers consist of a pair of real numbers of identical precision. Each real or complex number is associated with a corresponding field structure, which has the same precision as all of its elements. Magma stores a list of real and complex fields that have been created during a session, and it is guaranteed that any two fields of the same fixed precision are the same. This means in particular that changing the name of $\sqrt{-1}$ (see `AssignNames` below) on one of the complex fields of precision r will change the name on every complex field of that same precision. As a convenience, Magma allows real and complex numbers of differing precisions to be used in the same expression; internally, Magma implicitly reduces the precision of the higher precision element to the precision of the lower element.

While internally we store real numbers in base two, when creating real or complex fields the precision is by default specified in the number of *decimal* digits, not binary digits, required. It is possible to specify the precision in binary digits if needed see the documentation for *RealField* for details).

Example: Real Numbers and Loss of Precision (ex-5afbbc)

We show how to create and manipulate real numbers. In particular, note that there is an inherent loss of precision in the conversion between base 10 and base 2 representations of some real numbers.

```
> S1 := RealField(20);
> S2 := RealField(10);
> a := S1 ! 0.5;
> a;
0.50000000000000000000
> b := S2 ! 0.05;
> b;
0.05000000000
> a + b;
0.5500000000
> Precision(a + b);
10
```

A warning is in place here; in the examples above, the real number on the right hand side had to be constructed in some real field *before* it could be coerced into S_1 and S_2. That real field is the so-called *default real field*. In these examples it is assumed that the default field has sufficiently large precision to store the real numbers on the right accurately to the last digit.

26.1.2 Coercion

Automatic coercion ensures that all functions listed below that take an element of some real field as an argument, will also accept an integer or a rational number as an argument; in this case the integer or rational number will be coerced automatically into the default real field. For the binary operations (such as `+, *`) coercion also takes place: if one argument is real and the other is integral or rational, automatic coercion will put them both in the parent field of the real argument. If the arguments are real numbers of different fixed precision, the result will have the smaller precision of the two.

The same coercion rules apply for functions taking a complex number as an argument; in that case real numbers will be valid input as well: if necessary reals, rationals, or integers will be coerced into the appropriate complex field.

Elements of quadratic and cyclotomic fields that are real can be coerced into any real field using `!`; any quadratic or cyclotomic field element can be coerced by `!` into any complex field. Functions taking real or complex arguments will not *automatically* coerce such arguments though.

26.1.3 Homomorphisms

The only homomorphisms that have a real field or a complex field as domain are the coercion functions. Therefore, homomorphisms from the reals or complexes may be specified as follows.

```
hom< R -> S | >: FldRe, Str -> Map
```

Here S must be a structure into which all elements of the real or complex field R are coercible, such as another real or complex field, or a polynomial ring over one of these. These homomorphisms can also be obtained as map by using the function `Coercion`, also called `Bang`.

Example: Embedding a Real Field in a Polynomial Ring `(ex-edd37f)`

Here are two equivalent ways of creating the embedding function from a real field into a polynomial ring over some complex field.

```
> Re := RealField(20);
> PC<x, y> := PolynomialRing(ComplexField(8), 2);
> f := hom< Re -> PC | >;
> bangf := Bang(Re, PC);
> f(Pi(Re));
3.1415927
> f(Pi(Re)) eq bangf(Pi(Re));
true
```

26.1.4 Special Options

When Magma is started up, real and complex fields of precision 30 are created by default. They serve (among other things) as a parent for reals that are created as literals, such as 1.2345, in the same way as the default ring of integers is the parent for literal integers. It is possible to change this default real field with `SetDefaultRealField`.

Finally, `AssignNames` can be used to change the name for $\sqrt{-1}$ in a complex field.

`SetDefaultRealField(R): FldRe`

Procedure to change the default parent for literal real numbers to the real field R. This parent is the real field of precision 30 by default.

`GetDefaultRealField() -> FldRe`

Return the current parent for literal real numbers.

`AssignNames(~C, [s]): FldCom, [ MonStgElt ]`

Procedure to change the name of the purely imaginary element $\sqrt{-1}$ in the complex field C to the contents of the string s. When C is created, the name is "C.1"; suitable choices of s might be `"i"`, `"I"`, or `"j"`. This procedure only changes the name used in printing the elements of C. It does *not* assign to an identifier called s the value of $\sqrt{-1}$ in C; to do this, use an assignment statement, or use angle brackets when creating the field. Note that since this is a procedure that modifies C, it is necessary to have a reference `~C` to C in the call to this function.

`Name(C, 1): FldCom, RngIntElt -> FldComElt`

Given a complex field C, return the element which has the name attached to it, that is, return the purely imaginary element $\sqrt{-1}$ of C.

26.1.5 Version Functions

The following intrinsics retrieve the versions of MPFR, MPC, and GMP which the current Magma is using.

`GetGMPVersion()`

`GetMPFRVersion()`

`GetMPCVersion()`

The version of GMP, MPFR, or MPC being used.

26.2 Creation Functions

We describe the creation of real and complex fields and their elements.

26.2.1 Creation of Structures

At the time Magma is loaded, a real field is automatically created. This is used as the default parent for literal reals and real values returned by Magma.

`RealField(p): RngIntElt -> FldRe`

Bits: BoolElt	Default: false

Given a positive integer p, create and return a version R of the real field $\mathbb{R}$ in which all calculations are correct to precision p. If the parameter `Bits` is `true`, then the precision p is specified as the number of binary digits. If `Bits` is `false`, then the precision is given as the number of decimal digits — this is translated into a binary precision of $\lceil \log_2 10^p \rceil$.

`RealField() -> FldRe`

Return the default real field.

`ComplexField(p): RngIntElt -> FldCom`

```
Bits: BoolElt                    Default: false
```

Given a positive integer p, create and return a version C of the complex field $\mathbb{C}$ in which all calculations are correct to precision p. If the parameter `Bits` is `true`, then the precision p is specified as the number of binary digits. If `Bits` is `false`, then the precision is given as the number of decimal digits — this is translated into a binary precision of $\lceil \log_2 10^p \rceil$. By default no name is given to $\sqrt{-1}$; this may be changed with `AssignNames`. Angle brackets, e.g. `C<i> := ComplexField(20)`, may be used to assign $\sqrt{-1}$ to an identifier.

`ComplexField() -> FldCom`

Return the default complex field. By default no name is given to $\sqrt{-1}$; this may be changed with `AssignNames`. Angle brackets, e.g. `C<i> := ComplexField()`, may be used to assign $\sqrt{-1}$ to an identifier.

`ComplexField(R): FldRe -> FldCom`

Return the complex field which has real subfield R; in other words, return the complex field with the same precision as the real field R.

Example: Creating a Complex Field (ex-108c46)

It is convenient to use i to define elements of a complex field. It is also possible to change the default printing of i, using `AssignNames`, as follows. Note that the latter procedure does not assign to an identifier, it only changes the printing.

```
> C<i> := ComplexField(20);
> Pi(C)+ 1/4*i;
3.1415926535897932385 + 0.25000000000000000000*i
> AssignNames(~C, ["k"]);
> Pi(C)+ 1/4*i;
3.1415926535897932385 + 0.25000000000000000000*k
> k := Name(C, 1);
> Pi(C)+ 1/4*k;
3.1415926535897932385 + 0.25000000000000000000*k
```

26.2.2 Creation of Elements

`a . becpd: RngIntElt, RngIntElt, RngIntElt -> FldReElt`

`a . bEcPd: RngIntElt, RngIntElt, RngIntElt -> FldReElt`

`a.bec P d: RngIntElt, RngIntElt, RngIntElt -> FldReElt`

`a.bec p d: RngIntElt, RngIntElt, RngIntElt -> FldReElt`

`a.b E cpd: RngIntElt, RngIntElt, RngIntElt -> FldReElt`

`a.b e cpd: RngIntElt, RngIntElt, RngIntElt -> FldReElt`

Given a succession of literal decimal digits a, a succession of literal decimal digits b, a succession of literal decimal digits c, and an integer d, construct the real number $r = a.b \times 10^c$. If specified, the effect of d is to create r as an element of the real field of precision d. If d is omitted (together with `p` or `P`), the real number will be created as an element of the default real field. Both a and c may include a leading sign `+` or `-`; leading zeroes in a and c are ignored. If b consists entirely of zeroes it may be omitted together with the `.` and if c is zero it may be omitted together with `E` (or `e`). But note that if all of b, c, and d are omitted the result will be an integer.

`elt<R | m, n>: FldRe, FldReElt, RngIntElt -> FldReElt`

Given the real field R, an element m coercible into R, and an integer n, construct the real number $m \times 2^n$ in R.

`elt<C | x, y>: FldCom, FldReElt, FldReElt -> FldComElt`

`C ! [x, y]: FldCom, [FldReElt, FldReElt] -> FldComElt`

Given the complex field C and elements x and y coercible into the real field underlying C, construct the complex number $x + y$i.

`R ! a: FldRe, RngElt -> FldReElt`

Given an integer, a rational number, a quadratic or cyclotomic number field element a, this returns an element from the real field R that best approximates a. An error results if a is a non-real quadratic or cyclotomic field element. If R is a field of precision r and a is an element of a real field S of precision s then:

- if a is an element of a real field S of precision $s \geq r$, then an element of R approximating a to r digits is returned;
- if a is an element of a real field S of precision $s < r$, then an element of R is returned approximating a, obtained by padding with zeroes until the required precision r is reached;

`C ! a: FldCom, RngElt -> FldComElt`

Given an integer, a rational number, a quadratic or cyclotomic number field element a, this returns an element from the complex field C that best approximates a. The rules of coercion for the real and imaginary parts are the same as those for coercion into a real field.

Example: Creating Real Numbers (ex-356721)

We create the real number 1.2345 in many ways. We assume that the default real field has not been changed.

```
> x := 1.2345;
> x, Parent(x);
1.234500000000000000000000000000 Real field of precision 30
> SetDefaultRealField(RealField(20));
> x1 := 1.2345;
> x1, Parent(x1);
1.2345000000000000000 Real field of precision 20
> x2 := 12345e-4;
> x2, Parent(x2);
1.2345000000000000000 Real field of precision 20
> x3 := 1.2345p10;
> x3, Parent(x3);
1.234500000 Real field of precision 10
> x4 := 12345e-4p8;
> x4, Parent(x4);
1.2345000 Real field of precision 8
> x5 := RealField(12) ! 1.2345;
> x5, Parent(x5);
1.23450000000 Real field of precision 12
```

The following generic element constructions are available; they return the 1 and 0 element of a real or complex field, where any zero elements are the "positive zero" in MPFR.

`One(R): FldRe -> FldReElt`

`One(R): FldCom -> FldComElt`

`Identity(R): FldRe -> FldReElt`

`Identity(R): FldCom -> FldComElt`

`Zero(R): FldRe -> FldReElt`

`Zero(R): FldCom -> FldComElt`

`Representative(R): FldRe -> FldReElt`

`Representative(R): FldCom -> FldComElt`

26.3 Structure Operations

26.3.1 Related Structures

`Category(R): FldRe -> Cat`

`Category(R): FldCom -> Cat`

`Parent(R): FldRe -> PowerStructure`

`Parent(R): FldCom -> PowerStructure`

`PrimeField(R): FldRe -> FldRat`

`PrimeField(R): FldCom -> FldRat`

26.3.2 Numerical Invariants

`Characteristic(R): FldRe -> RngIntElt`

`Characteristic(R): FldCom -> RngIntElt`

26.3.3 Ring Predicates and Booleans

`IsCommutative(R): FldRe -> BoolElt`

`IsCommutative(R): FldCom -> BoolElt`

`IsUnitary(R): FldRe -> BoolElt`

`IsUnitary(R): FldCom -> BoolElt`

`IsFinite(R): FldRe -> BoolElt`

`IsFinite(R): FldCom -> BoolElt`

`IsOrdered(R): FldRe -> BoolElt`

`IsOrdered(R): FldCom -> BoolElt`

`IsField(R): FldRe -> BoolElt`

`IsField(R): FldCom -> BoolElt`

`IsEuclideanDomain(R): FldRe -> BoolElt`

`IsEuclideanDomain(R): FldCom -> BoolElt`

`IsPID(R): FldRe -> BoolElt`

`IsPID(R): FldCom -> BoolElt`

`IsUFD(R): FldRe -> BoolElt`

`IsUFD(R): FldCom -> BoolElt`

`IsDivisionRing(R): FldRe -> BoolElt`

```
IsDivisionRing(R): FldCom -> BoolElt

IsEuclideanRing(R): FldRe -> BoolElt

IsEuclideanRing(R): FldCom -> BoolElt

IsPrincipalIdealRing(R): FldRe -> BoolElt

IsPrincipalIdealRing(R): FldCom -> BoolElt

IsDomain(R): FldRe -> BoolElt

IsDomain(R): FldCom -> BoolElt

R eq S: FldRe, FldRe -> BoolElt

R ne S: FldRe, FldRe -> BoolElt

R eq S: FldCom, FldCom -> BoolElt

R ne S: FldCom, FldCom -> BoolElt
```

26.3.4 Other Structure Functions

`Precision(R): FldCom -> RngIntElt`
`Precision(R): FldRe -> RngIntElt`

Return the decimal precision p to which calculations are performed in the real or complex field R.

`BitPrecision(R): FldCom -> RngIntElt`
`BitPrecision(R): FldRe -> RngIntElt`

Return the (internally used) bit precision p to which calculations are performed in the real or complex field R.

26.4 Element Operations

26.4.1 Generic Element Functions and Predicates

All predicates on real or complex numbers that check whether these numbers are equal to an integer do so within the given precision of the parent field. Thus `IsOne(c)` for an element of a complex domain of precision 20 returns true if and only if the real part equals one and the imaginary part equals 0 up to 20 decimals.

Note that `IsZero` will match either the positive or negative zero in MPFR.

```
Parent(r): FldReElt -> Rng

Parent(r): FldComElt -> Rng

Category(r): FldReElt -> Cat

Category(r): FldComElt -> Cat

IsZero(r): FldReElt -> BoolElt

IsZero(r): FldComElt -> BoolElt

IsOne(r): FldReElt -> BoolElt

IsOne(r): FldComElt -> BoolElt

IsMinusOne(r): FldReElt -> BoolElt
```

`IsMinusOne(r): FldComElt -> BoolElt`

`IsUnit(r): FldReElt -> BoolElt`

`IsUnit(r): FldComElt -> BoolElt`

`IsZeroDivisor(r): FldReElt -> BoolElt`

`IsZeroDivisor(r): FldComElt -> BoolElt`

`IsIdempotent(r): FldReElt -> BoolElt`

`IsIdempotent(r): FldComElt -> BoolElt`

`IsNilpotent(r): FldReElt -> BoolElt`

`IsNilpotent(r): FldComElt -> BoolElt`

`IsIrreducible(r): FldReElt -> BoolElt`

`IsIrreducible(r): FldComElt -> BoolElt`

`IsPrime(r): FldReElt -> BoolElt`

`IsPrime(r): FldComElt -> BoolElt`

26.4.2 Comparison of and Membership

The (in)equality test on real numbers of only test for equality up to the given precision. Equality testing on complex numbers is done by testing the real and imaginary parts.

Again the positive and negative zero in MPFR are equal.

The comparison functions `gt, ge, lt, le` are not defined for complex numbers.

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

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

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

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

`a in R: FldReElt, FldRe -> BoolElt`

`a notin R: FldReElt, FldRe -> BoolElt`

`a in R: FldComElt, FldCom -> BoolElt`

`a notin R: FldComElt, FldCom -> BoolElt`

`a gt b: FldReElt, FldReElt -> BoolElt`

`a ge b: FldReElt, FldReElt -> BoolElt`

`a lt b: FldReElt, FldReElt -> BoolElt`

`a le b: FldReElt, FldReElt -> BoolElt`

`Maximum(a, b): FldReElt, FldReElt -> FldReElt`

`Minimum(a, b): FldReElt, FldReElt -> FldReElt`

`Maximum(Q): [FldReElt] -> FldReElt`

`Minimum(Q): [FldReElt] -> FldReElt`

26.4.3 Other Predicates

`IsIntegral(c): FldReElt -> BoolElt`

`IsIntegral(c): FldComElt -> BoolElt`

Returns `true` if and only if the real or complex number c is a rational integer.

`IsReal(c): FldComElt -> BoolElt`

Returns `true` if the complex number c is real, `false` otherwise. This checks whether the digits of the imaginary part of c are 0 up to the precision of the parent complex field.

26.4.4 Arithmetic

The binary operations `+, -, *, /` allow combinations of arguments from the integers, the rationals, and real and complex fields; automatic coercion is applied where necessary (see the Introduction).

```
+ r: FldReElt -> FldReElt

+ r: FldComElt -> FldComElt

- r: FldReElt -> FldReElt

- r: FldComElt -> FldComElt

r + s: FldReElt, FldReElt -> FldReElt

r + s: FldComElt, FldComElt -> FldComElt

r - s: FldReElt, FldReElt -> FldReElt

r - s: FldComElt, FldComElt -> FldComElt

r * s: FldReElt, FldReElt -> FldReElt

r * s: FldComElt, FldComElt -> FldComElt

r / s: FldReElt, FldReElt -> FldReElt

r / s: FldComElt, FldComElt -> FldComElt

r ^ k: FldReElt, FldReElt -> FldReElt

r ^ k: FldReElt, RngIntElt -> FldReElt

r ^ k: FldComElt, RngIntElt -> FldComElt

r ^ k: FldComElt, FldComElt -> FldComElt

r +:= s: FldReElt, FldReElt -> FldReElt

r +:= s: FldComElt, FldComElt -> FldComElt

r -:= s: FldReElt, FldReElt -> FldReElt

r -:= s: FldComElt, FldComElt -> FldComElt

r *:= s: FldReElt, FldReElt -> FldReElt

r *:= s: FldComElt, FldComElt -> FldComElt

r /:= s: FldReElt, FldReElt -> FldReElt

r /:= s: FldComElt, FldComElt -> FldComElt

r ^:= s: FldReElt, FldReElt -> FldReElt

r ^:= s: FldComElt, FldComElt -> FldComElt
```

26.4.5 Conversions

Here we list various ways to convert between integers, reals of fixed precision, complexes, and their various representations, other than by the creation functions and `!`. See also the rounding functions in a later section.

`MantissaExponent(r): FldReElt -> RngIntElt, RngIntElt`

Given a real number r, this function returns an integer m (the mantissa of r) and an integer e (the exponent of r) such that $r \approx m \times 2^e$, with the size of m being the same as that of the internal representation of r.

`ComplexToPolar(c): FldComElt -> FldReElt, FldReElt`

Given a complex number c, return the modulus $m \geq 0$ and the argument a (with $-\pi \leq a \leq \pi$) of c as real numbers to the same precision as c.

`PolarToComplex(m, a): FldReElt, FldReElt -> FldComElt`

Given real numbers m and a, construct the complex number me^{ia}. The result will have the smaller of the precisions of m and a; each of m and a is allowed to be an integer or rational number; if both are integral or rational then the result will have the default precision, otherwise the result will be of the same precision as the real argument.

`Argument(c): FldComElt -> FldReElt`

`Arg(c): FldComElt -> FldReElt`

Given a complex number c, return the real number (to the same precision) that is the argument (in radians between $-\pi$ and π) of c.

Example: Argument of a Signed Zero `(ex-9ff75f)`

We note the argument of -1 with an imaginary part of positive zero differs from that of the argument of -1 with an imaginary part of negative zero. This follows the MPFR/MPC specifications.

```
> C := ComplexField(30);
> x := -One(C);
> Imaginary(x); // imaginary part is negative zero
-0.00000000000000000000000000000
> Arg(x); // approximately negative pi as Arg
-3.1415926535897932384626433832
> y := C!-1; // coerce, so imag part is positive zero
> Imaginary(y); // positive zero
0.00000000000000000000000000000
> Arg(y); // approximately positive pi as Arg
3.1415926535897932384626433832
> assert x eq y; // note that x and y are indeed equal!
```

`Modulus(c): FldComElt -> FldReElt`

Given a complex number c, return the real number (to the same precision as c) that is the modulus of c.

`Real(c): FldComElt -> FldReElt`

`Re(c): FldComElt -> FldReElt`

Given a complex number $c = x + y\mathrm{i}$, return the real part x of c (as a real number to the same precision as c).

`Imaginary(c): FldComElt -> FldReElt`

`Im(c): FldComElt -> FldReElt`

Given a complex number $c = x + y\mathrm{i}$, return the imaginary part y of c (as a real number to the same precision as c).

26.4.6 Rounding

`Round(r): FldReElt -> FldReElt`

`Round(r): FldComElt -> RngQuadElt`

Given a real number r, return the integer i for which $|r - i|$ is a minimum. i.e., the integer closest to r. If there are two such integers, the one of larger magnitude is chosen (rounding away from zero). Given a (non-real) complex number r, return the Gaussian integer i for which $|r - i|$ is a minimum, i.e. the Gaussian integer closest to r.

`Truncate(r): FldReElt -> RngIntElt`

Given a real number r, return $\lfloor r \rfloor$ if r is positive, and return $-\lfloor -r \rfloor + 1$ if r is negative. Thus, the effect of this function is to round towards zero.

`Ceiling(r): Infty -> Infty`

`Ceiling(r): FldReElt -> RngIntElt`

The ceiling of the real number r, i.e. the smallest integer greater than or equal to r.

`Floor(r): Infty -> Infty`

`Floor(r): FldReElt -> RngIntElt`

The floor of the real number r, i.e. the greatest integer less than or equal to r.

26.4.7 Precision

`Precision(r): FldReElt -> RngIntElt`

`Precision(c): FldComElt -> RngIntElt`

Given a real or complex number c belonging to the real or complex field C, return the decimal precision p to which calculations are performed in C.

`BitPrecision(r): FldReElt -> RngIntElt`

`BitPrecision(c): FldComElt -> RngIntElt`

Given a real or complex number c belonging to the real or complex field C, return the (internally used) bit precision p to which calculations are performed in C.

`Precision(L): [FldReElt] -> RngIntElt`

`Precision(L): [FldComElt] -> RngIntElt`

Gives a sequence of real or complex numbers, return the precision p of their parent field.

`ChangePrecision(r, n): FldReElt, RngIntElt -> FldReElt`

`ChangePrecision(c, n): FldComElt, RngIntElt -> FldComElt`

Coerces the real (r) or complex (c) number into a field of precision n.

26.4.8 Constants

Let R denote a real or complex field. The functions described below will return an approximation of certain constants to the precision associated with a given real or complex field R. If R is real, a real number is returned; if R is complex, a complex number with imaginary part zero is returned.

`Catalan(R): FldRe -> FldReElt`

`Catalan(R): FldCom -> FldComElt`

The value of Catalan's constant computed to the accuracy associated with the real or complex field R. Catalan's constant is the sum

$$\sum_{k=0}^{\infty} (-1)^k (2k+1)^{-2}.$$

MPFR calculates this constant using formula (31) of Victor Adamchik's document "33 representations for Catalan's constant" `https://web.archive.org/web/20120805142652/http://www.cs.cmu.edu/~adamchik/articles/catalan/catalan.htm`, for more information see `mpfr.org`.

`EulerGamma(R): FldRe -> FldReElt`

EulerGamma(R): FldCom -> FldComElt

The value of Euler's constant

$$\gamma = \lim_{n\to\infty}(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} - \log n) \approx 0.57721566$$

computed to the precision of R.

Pi(R): FldRe -> FldReElt

Pi(R): FldCom -> FldComElt

The value of π computed to the precision of R.

26.4.9 Simple Element Functions

AbsoluteValue(r): FldReElt -> FldReElt

AbsoluteValue(r): FldComElt -> FldReElt

Abs(r): FldReElt -> FldReElt

Abs(r): FldComElt -> FldReElt

The absolute value of the real or complex number r.

Sign(r): FldReElt -> RngIntElt

Return one of the integer values $+1$, 0, -1 depending upon whether the real number r is positive, zero, or negative, respectively.

ComplexConjugate(r): FldReElt -> FldReElt

ComplexConjugate(c): FldComElt -> FldComElt

Conjugate(c): FldComElt -> FldComElt

The complex conjugate $x - y\mathrm{i}$ of a complex number $x + y\mathrm{i}$.

Norm(c): FldComElt -> FldReElt

Norm(r): FldReElt -> FldReElt

The real norm of a real or complex number c; note that for complex $c = x + y\mathrm{i}$ this returns $x^2 + y^2$, while for elements of real domains it just returns the absolute value. The result lies in the same field as the argument.

Root(r, n): FldReElt, RngIntElt -> FldReElt

Root(r, n): FldComElt, RngIntElt -> FldComElt

Given a real number R and a positive integer n, calculate $\sqrt[n]{r}$ (using Newton's method without divisions) with the same precision. If n is even then r must be non-negative.

SquareRoot(c): FldComElt -> FldComElt

Sqrt(c): FldComElt -> FldComElt

SquareRoot(r): FldReElt -> FldReElt

Sqrt(r): FldReElt -> FldReElt

Given a real or complex number c, return the square root of r as an element of the same field to which r belongs.

Distance(x, L): FldReElt, [FldReElt] -> FldReElt, RngIntElt

Distance(x, L): FldReElt, [FldComElt] -> FldReElt, RngIntElt

Distance(x, L): FldComElt, [FldReElt] -> FldReElt, RngIntElt

Distance(x, L): FldComElt, [FldComElt] -> FldReElt, RngIntElt

Max: ExtReElt	Default: Infinity()

Given a sequence L of real or complex numbers and an additional number x compute the distance between x and L, ie. $\min_{y\in L}|x - y|$, that is the shortest distance between x and any element of L. Furthermore, the index in L of an element realising the distance is returned as a second argument.

Note: prior to V2.21, the `Distance` was only to be used for *short* distances, and thus a maximum of 1 was returned. Through the use of a new `Max` vararg (setting this equal to 1), this old behaviour can be obtained.

`Diameter(L): [FldReElt] -> FldReElt`

`Diameter(L): [FldComElt] -> FldReElt`

`Max: ExtReElt` `Default: Infinity()`

Given a sequence L of real of complex numbers, compute the diameter of the set defined by L, ie. the smallest distance between distinct elements of L.

Note: prior to V2.21, the `Diameter` was only to be used for *short* distances, and thus a maximum of 1 was returned. Through the use of a new `Max` vararg (setting this equal to 1), this old behaviour can be obtained.

26.4.10 Roots

Magma contains a very powerful algorithm for finding highly accurate approximations to the complex roots of a polynomial; it is based on Xavier Gourdon's implementation of Schönhage's algorithm, which we will summarize below.

Given a polynomial $p = a_0 + a_1 z + \cdots + a_n z^n \in \mathbb{C}[z]$, define the norm of p, $|p|$, by

$$|p| = |a_0| + |a_1| + \cdots + |a_n|.$$

Schönhage's algorithm (given in his technical report of 1982 [Sch82]) takes as input a univariate polynomial p in $\mathbb{C}[z]$ and a positive real number ε, and finds linear factors $L_j = u_j z - v_j$ $(j = 1, \ldots, n = \deg(p))$ such that

$$|p - L_1 \cdots L_n| < \varepsilon |p|.$$

The parameter ε may be chosen so as to find the roots of p to within a certain ε', and this is how the function `Roots` described below works (when run with Schönhage's algorithm).

The algorithm uses the concept of a 'splitting circle' to find polynomials F and G such that $|p - FG| < \varepsilon_1 |p|$ for some ε_1 depending on ε.

This splitting circle method can then be applied recursively to F and G until we have only linear factors, as required.

The splitting circle method works as follows. For the purposes of this discussion assume that p is monic. Suppose we know a circle Γ such that, for some integer k with $0 < k < n$, there are k roots of p (say $u_1, \ldots, u_k$) which lie inside Γ, and the other $n - k$ roots $(u_{k+1}, \ldots, u_n)$ lie outside Γ. Note that the circle Γ is chosen so that the roots of p are not too close to it. Then we can write $p = FG$, where $F = (z - u_1) \ldots (z - u_k)$ and $G = (z - u_{k+1}) \ldots (z - u_n)$. Through shifts and scalings, we may assume that $\Gamma = \{c \in \mathbb{C} : \|z\| = 1\}$.

For m in $\{1, \ldots, k\}$, let s_m denote the m-th power sum of the roots of p which lie inside the splitting circle. That is,

$$s_m = u_1^m + \cdots + u_k^m.$$

The residue theorem can then be used to calculate s_m $(1 \leq m \leq k)$:

$$s_m = \frac{1}{2\pi i} \int_\Gamma z^m \frac{p'(z)}{p(z)}\, dz.$$

where the integration can be computed to the required precision by the discrete sum

$$s_m \approx \frac{1}{N} \sum_{j=0}^{N-1} \frac{p'(\omega^j)}{p(\omega^j)} \omega^{(m+1)j}.$$

for a large enough integer N, where $\omega = \exp(2\pi i/N)$.

The coefficients of the polynomial F can then be computed from the Newton sums s_m $(1 \le m \le k)$ using the classical Newton formulae. Then set $G = p/F$.

The integer N above needed to get F and G to the required precision can be quite large. It is more efficient to use a smaller value of N to give an approximation F_0 of F, and then use the following refining technique.

Define G_0 (an approximation of G) by $p = F_0G_0 + r$, where $\deg(r) < \deg(F_0)$. We want polynomials f and g such that $F_1 = F_0 + f$ and $G_1 = G_0 + g$ are better approximations of F and G. Now

$$p - F_1G_1 = p - F_0G_0 - fG_0 - gF_0 - fg.$$

Hence choosing f and g such that

$$p - F_0G_0 = fG_0 + gF_0$$

will lead to a second order error.

The Euclidean algorithm could be used to find f and g, but this is numerically unstable. It suffices to find polynomials H (called the auxiliary polynomial) and L such that $1 = HG_0 + LF_0$, where $\deg(H) < \deg(F_0)$ and $\deg(L) < \deg(G_0)$.

The polynomial H can be calculated using the formula

$$H(z) = \frac{1}{2\pi i} \int_\Gamma \frac{1}{(F_0G_0)(t)} \frac{F_0(z) - F_0(t)}{z - t}\, dt.$$

Again, rather than computing the integral to the required precision directly, we find only an approximation H_0 and then refine it using Newton iteration:

$$H_{m+1} \equiv H_m(2 - H_mG_0) \pmod{F_0}.$$

Assuming that $|H - H_0|$ is small, the sequence (H_m) converges quadratically to H.

Once H is known to a large enough precision, f can be computed by

$$f \equiv H(p - F_0G_0) \pmod{F_0}.$$

This gives us the new approximation F_1 of F, and G_1 is computed by division of p by F_1. We repeat this process until

$$|p - FG| < \varepsilon_1|p|$$

and we are done.

The problem remains to find the splitting circle.

This relies mainly on the computation of the moduli of the roots of p. Let $r_1(p) \le r_2(p) \le \cdots \le r_n(p)$ denote the moduli of the roots of p in ascending order. For each k, the computation of $r_k(p)$ with a small number of digits can be achieved in a reliable way using the Graeffe process. The Graeffe process is a root squaring step transforming any given polynomial p into a polynomial q of the same degree whose roots are the square of the roots of p.

By the use of a suitable shift, we may assume that the sum of the roots of p is zero. If $p(0) = 0$, then we have found a factorization $p \approx FG$ with $F = z$ and $G = p/z$. If not, then the computation of the maximum root modulus $r_n(p)$ allows us to scale p so that its maximum root modulus is now close to 1. For $j = 0, 1, 2, 3$, set

$$q_j(z) = p(z + 2i^j).$$

Then amongst these four polynomials there exists q such that

$$\frac{r_n(q)}{r_1(q)} = \exp(\Delta),$$

with $\Delta > 0.3$. A dichotomic process from the computation of some $r_j(q)$ can then be applied to find k $(1 \leq k \leq n-1)$ such that

$$\frac{r_{k+1}(q)}{r_k(q)} > \exp(\frac{\Delta}{n-1}).$$

Then the circle $\{c : \|c\| = \sqrt{r_k(q)r_{k+1}(q)}\}$ is a suitable splitting circle, with the roots not too close to it.

`Roots(p): RngUPolElt -> [ <FldComElt, RngIntElt> ]`

```
Al    : MonStgElt                    Default: "Schonhage"
Digits: RngIntElt                    Default:
```

Given a univariate polynomial p over a real or complex field, this returns a sequence of complex approximations to the roots of p. The elements of this sequence are of the form $< r, m >$, where r is a root and m its multiplicity.

The algorithm used to find the roots of p may be specified by using the optional argument `Al`. This must be one of `"Schonhage"` (which is the default), `"Laguerre"`, `"NewtonRaphson"`, or `"Combination"` (a combination of Laguerre and Newton-Raphson). When using the (default) Schönhage algorithm, the roots given are correct to within an absolute error of 10^{-d}, where d is the value of `Digits`. This algorithm gives correct results in all cases. When using the other algorithms (for which correct answers are not guaranteed in all cases), the results are found with `Digits` significant figures. The default value for `Digits` is the current precision of the free real field.

Pari is used here for complex polynomials.

Warning: Beware of the problems of floating point numbers. Because real numbers are stored in the computer with finite precision, you may not be finding the roots of the polynomial you want. If you know the polynomial exactly, you should enter it with exact (that is, integer or rational) coefficients. This is illustrated in the following example.

Example: Roots of a Polynomial with a Multiple Root (ex-04bda5)

```
> P<z> := PolynomialRing(ComplexField());
> p := (z-1.1)^6;
> p;
z^6 - 6.60000000000000000000000000001*z^5 +
    18.150000000000000000000000000*z^4 -
    26.620000000000000000000000000*z^3 +
    21.961500000000000000000000000*z^2 -
    9.663060000000000000000000003*z +
    1.771561000000000000000000001
> R := Roots(p);
> R;
[ <1.100013305965906054216519998 57, 1>,
<1.100006652894308609692986689 17 +
1.152330449581798256514866512 57E-5*i, 1>,
<1.100006652894308609692986689 17 -
1.152330449581798256514866512 57E-5*i, 1>,
<1.099986694211380305218347312 34, 1>,
<1.099993347017048210589579655 37 +
1.152315096132698580046691677 11E-5*i, 1>,
<1.099993347017048210589579655 37 -
1.152315096132698580046691677 11E-5*i, 1> ]
> P<x> := PolynomialRing(Rationals());
> q := (x-11/10)^6;
> Roots(q);
[ <11/10, 6> ]
```

The function RootsNonExact (below) is more suitable for non-exact polynomials.

RootsNonExact(p): RngUPolElt[FldRe] -> [FldComElt], [FldComElt]

RootsNonExact(p): RngUPolElt[FldCom] -> [FldComElt], [FldComElt]

Given a polynomial p of degree n defined over a real or complex field, returns a sequence $[v_1, \ldots, v_n]$ of complex numbers such that

$$|p - a(z - v_1) \ldots (z - v_n)| < 10^{-d}|p|,$$

where a is the leading coefficient of p, and d is the precision of the field.

A second sequence $[e_1, \ldots, e_n]$ of (free) real numbers may also be returned. Given any polynomial $\hat{p}$ such that $|p - \hat{p}| < 10^{-d}|p|$, we can write $\hat{p} = a(z - u_1) \ldots (z - u_n)$ with $|v_i - u_i| < e_i$. In some cases, such error bounds cannot be derived, because the value d of Digits is too small for the given polynomial. In these cases, this second sequence is not returned.

This function acknowledges the fact that the polynomial p may not be the exact polynomial wanted, but only an approximation (to a certain number of decimal places), and so the roots of the true polynomial can only be found to a limited number of decimal places. Increasing the precision will decrease the errors on the 'roots'.

Example: Roots with Error Estimates (ex-a1670d)

```
> P<z> := PolynomialRing(ComplexField());
> p := (z-1.1)^6;
> R, E := RootsNonExact(p);
> R;
[ 1.100014832969134514103701910 06 -
8.56404454142796527111307103383E-6*i,
1.100014832969134514103701910 06 +
8.56404454142796527111304767692E-6*i,
1.099985167421993219043939759 59 +
8.56336708832137724325720239457E-6*i,
1.099999999608872266852358330 26 +
1.71274116266516824685125851069E-5*i,
1.099985167421993219043937716 23 -
8.56336708832137723945115457519E-6*i,
1.099999999608872266852360373 65 -
1.71274116266516824723187272572E-5*i ]
> E;
[ 0.004823144154215697199106216430 66,
0.004823144154215697199106216430 66,
0.004823014089197386056184768676 76,
0.004823079112611594609916210174 56,
0.004823014089197386056184768676 76,
0.004823079112611594609916210174 56 ]
```

HenselLift(f, R, k): RngUPolElt, FldReElt, RngIntElt -> FldReElt

HenselLift(f, R, k): RngUPolElt, FldComElt, RngIntElt -> FldComElt

Let f be a real or complex polynomial and x an approximation to a single zero of f. This function will apply the Newton-iteration to improve the accuracy of the root to the precision indicated by k.

26.4.11 Continued Fractions

The following functions use the continued fraction expansion of real numbers to get Diophantine approximations. They were obtained from corresponding Pari implementations.

ContinuedFraction(r): FldRatElt -> [RngIntElt]

`ContinuedFraction(r): FldReElt -> [ RngIntElt ]`

`Bound: RngIntElt`	`Default: -1`

Given an element r from a real field, return a sequence of integers s that form the partial quotient for the (regular) continued fraction expansion for r, so r is approximately equal to

$$s_1 + \frac{1}{s_2 + \frac{1}{s_3 + \cdots + \frac{1}{s_n}}}.$$

The length n of the sequence is determined in such a way that the last significant partial quotient is obtained (determined by the precision with which r is known), unless the optional integer argument `Bound` is used to limit the length.

`BestApproximation(r, n): FldReElt, RngIntElt -> FldReElt`

Given an element r from a real field and a positive integer n, this function determines a rational approximation to r with denominator not exceeding n. The approximation is at least as close as the best continued fraction convergent with denominator not exceeding n. Pari is used here.

`Convergents(s): [ RngIntElt ] -> ModMatRngElt`

Given a sequence s of n non-negative integers (forming the partial fractions of a real number r, say), this function returns a 2×2 matrix with integer coefficients

$$\begin{pmatrix} p_n & p_{n-1} \\ q_n & q_{n-1} \end{pmatrix};$$

the quotients p_{n-1}/q_{n-1} and p_n/q_n form the last two convergents for r as provided by s.

26.4.12 Linear and Algebraic Dependencies

`LinearRelation(q: parameters): [ FldComElt ] -> [ RngIntElt ]`

`LinearRelation(v: parameters): ModTupRngElt -> ModTupRngElt`

`Al: MonStgElt`	`Default: "LLL"`

(Deprecated). Given a sequence q or a vector v with entries from a complex field, return an integer sequence or vector forming the coefficients for a (small) linear dependency among the entries. The algorithm now only uses `"LLL"` (not "Hastad"), and this function is deprecated in any case. The new version of this function is `IntegerRelation`.

`AllLinearRelations(q, p): SeqEnum, RngIntElt -> Lat`

Given a sequence q with entries from a real or complex field, return the lattice of all (small) integer linear dependencies among the entries. The precision p is used both in the relation-detection, and to define "small" dependencies.

`IntegerRelation(q): SeqEnum -> SeqEnum, FldReElt`

`IntegerRelation(q, N): SeqEnum, RngIntElt -> SeqEnum, FldReElt`

`Delta: FldReElt`	`Default: 0.75`

Given a sequence $q = (q_1, \ldots, q_d)$ with entries from a real or complex field, along possibly with an integer $N > 0$, return a sequence $(n_1, \ldots, n_d)$ of integers such that: any n_i has magnitude less or equal to N and the quantity $|\sum n_i q_i|$ is small. The routine is based on the LLL lattice reduction algorithm [LLL82] (see Chapter Reduction of Matrices and Lattices). The second return value is a measure of goodness-of-fit, and the second input value will be determined from the input precision of real/complex numbers if it is omitted.

`PowerRelation(r, k: parameters): FldReElt, RngIntElt -> RngUPolElt`

`PowerRelation(r, k: parameters): FldComElt, RngIntElt -> RngUPolElt`

```
Al       : MonStgElt                    Default: "LLL"
Precision: RngIntElt                    Default:
```

(Deprecated). Given an element r from a real or complex field, and an integer $k > 0$, return a univariate integer polynomial of degree at most k having r as an approximate root. The parameters here have the same usage and meaning as for `LinearRelation`. Pari is used here. The new version of this function is `MinimalPolynomial`.

`MinimalPolynomial(r, d, N): FldReElt, RngIntElt, RngIntElt -> RngUPolElt, FldReElt`

`MinimalPolynomial(r, d, N): FldComElt, RngIntElt, RngIntElt -> RngUPolElt, FldReElt`

`MinimalPolynomial(r, d): FldReElt, RngIntElt -> RngUPolElt -> FldReElt`

`MinimalPolynomial(r, d): FldComElt, RngIntElt -> RngUPolElt -> FldReElt`

```
Delta      : FldReElt                   Default: 0.75
ExactDegree: BoolElt                    Default: false
Squarefree : BoolElt                    Default: true
```

Given an element r from a real or complex field and two integers $d \geq 1$ and $N \geq 1$, tries to compute a univariate integer polynomial of degree at most d and coefficients of absolute values no greater than N which is the minimal polynomial of a number close to r. The routine relies on the LLL lattice reduction algorithm [LLL82] (see Chapter Reduction of Matrices and Lattices). When N is not given, it is determined from the precision of r. The `ExactDegree` vararg requires that the output degree be exactly as stated, and `Squarefree` removes any repeated factors. The second return value is a measure of goodness-of-fit.

Example: Polynomial Factorization via LLL `(ex-b60a49)`

This example illustrates the `MinimalPolynomial` routine. It describes informally the Kannan-Lenstra-Lovász algorithm to factor univariate integer polynomials [KLL84]. The idea is to compute a complex root of the polynomial P to be factored, and then compute the minimal polynomial Q of this root. This polynomial is irreducible and divides P. We then continue recursively with P/Q.

Notice that the polynomial factorisation routine `Factorization` (see Section *Factorization*) relies upon Van Hoeij's algorithm [vH01, vH02]. Though both approaches have polynomial time complexities, Van Hoeij's algorithm is significantly faster.

```
> C<i> := ComplexField(1000);
> P<X> := PolynomialRing(Integers());
> P_C<Z> := PolynomialRing(C);
> p0 := 5*X^12 + 64*X^11 + 51*X^10 + 38*X^9 - 60*X^8 - 56*X^7
>     - 51*X^6 - 14*X^5 - X^4 + 12*X^3 + 8*X^2 + 4*X;
> R := Roots (P_C!p0);
> root := R[1][1];
> res0 := MinimalPolynomial (root, 12, 100);
> p1 := P!(p0/res0);
> //
> R := Roots (P_C!p1);
> root := R[1][1];
> res1 := MinimalPolynomial (root, 11, 100);
> p2 := P!(p1/res1);
> //
> R := Roots (P_C!p2);
> root := R[1][1];
> res2 := MinimalPolynomial (root, 5, 100);
> p3 := P!(p2/res2);
> //
> R := Roots (P_C!p3);
> root := R[1][1];
```

(continues on next page)

(continued from previous page)

```
> res3 := MinimalPolynomial (root, 4, 100);
> //
> assert p0 eq res0*res1*res2*res3;
> res0, res1, res2, res3;
X
X^6 + 13*X^5 + 13*X^4 + 13*X^3 - 4*X^2 - 4*X - 4
X - 1
5*X^4 + 4*X^3 + 3*X^2 + 2*X + 1
```

26.5 Transcendental Functions

26.5.1 Exponential, Logarithmic, and Polylogarithmic Functions

In this section the exponential and logarithmic functions to the natural base e are described, as well as the conversion to the logarithm with respect to any base. The power series expansions are

$$e^z = \sum_{n=0}^{\infty} \frac{z^n}{n!}, \quad \ln(1+z) = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{z^n}{n}.$$

Further information on the Dilog and Polylog functions can be found in Lewin [Lew81].

Exp(f): RngSerElt -> RngSerElt

Given a power series f defined over a real or complex field, return the exponential power series of f.

Exp(c): FldComElt -> FldComElt

Exp(r): FldReElt -> FldReElt

Given an arbitrary real or complex number c, return the exponential e^c of c. Here c is allowed to be free or of fixed precision, and the result is in the same field as c.

Log(f): RngSerElt -> RngSerElt

Given a power series f defined over a real or complex field, return the logarithm of f. The valuation of f must be zero.

Log(r): FldReElt -> FldReElt

Log(c): FldComElt -> FldComElt

Given a non-zero real or complex number c, return the logarithm of c (to the natural base e). The principal value with imaginary part in $(-\pi, \pi]$ is chosen. The result will be a complex number, unless the argument is real and positive, in which case a real number is returned.

Log(b, r): FldReElt -> FldReElt

Given real numbers $b > 0$, $b \neq 1$, and $r > 0$, return the logarithm $\log_b(r)$ of r to the base b. Automatic coercion is applied if necessary.

Dilog(s): FldComElt -> FldComElt

Dilog(s): FldReElt -> FldReElt

For a given complex s, this returns the value of the principal branch of the *dilogarithm* $\mathrm{Li}_2(s)$, which can be defined by

$$\mathrm{Li}_2(s) = -\int_0^s \frac{\log(1-s)}{s} ds,$$

and forms the analytic continuation of the power series

$$\sum_{n=1}^{\infty} \frac{s^n}{n^2},$$

(which is convergent for $|s| \leq 1$). For large values of the argument a functional equation like

$$\mathrm{Li}_2(\frac{-1}{s}) + \mathrm{Li}_2(-s) = 2\mathrm{Li}_2(-1) - \frac{1}{2}\log^2(s)$$

should be used.

`Polylog(m, f): RngIntElt, RngSerElt -> RngSerElt`

For an integer $m \geq 2$ and power series f defined over a real or complex field, return the m-th polylogarithm of the series f. The valuation of f must be positive for $m > 1$.

`Polylog(m, s): RngIntElt, FldComElt -> FldComElt`

For given integer $m \geq 2$ and complex s this returns the value of the principal branch of the *polylogarithm* $\mathrm{Li}_m(s)$, defined for $m \geq 3$ by

$$\mathrm{Li}_m(s) = \int_0^s \frac{\mathrm{Li}_{m-1}(s)}{s} ds$$

(and for $m = 2$ as the dilogarithm Li_2). Then Li_m is the analytic continuation of

$$\sum_{n=1}^{\infty} \frac{s^n}{n^m},$$

(which is convergent for $|s| \leq 1$). For large values of the argument a functional equation like

$$(-1)^m \mathrm{Li}_m(\frac{-1}{s}) + \mathrm{Li}_m(-s) = -\frac{1}{m!}\log^m(s) + 2\sum_{r=1}^{\lfloor m/2 \rfloor} \frac{\log^{m-2r}(s)}{(m-2r)!}\mathrm{Li}_{2r}(-1)$$

should be used. Pari is used here.

`PolylogD(m, s): RngIntElt, FldComElt -> FldComElt`

`PolylogDold(m, s): RngIntElt, FldComElt -> FldComElt`

`PolylogP(m, s): RngIntElt, FldComElt -> FldComElt`

Given integer $m \geq 2$ and complex s, this returns the value of the principal branch of the modified versions $\tilde{D}_m, D_m$ and P_m of the polylogarithm $\mathrm{Li}_m(s)$; all of these satisfy functional equations of the form $f_m(1/s) = (-1)^m f_m(s)$. For their definition and main properties, see Zagier [Zag]. Pari is used here.

26.5.2 Trigonometric Functions

The trigonometric functions may be computed for real and complex arguments or for power series defined over a real or complex field. The basic power series expansions are

$$\sin(z) = \sum_{n=0}^{\infty} \frac{(-1)^{n+1} z^{2n+1}}{(2n+1)!}, \quad \cos(z) = \sum_{n=0}^{\infty} \frac{(-1)^n z^{2n}}{(2n)!}.$$

Euler's formulas relate these with the exponential functions via

$$\sin(z) = \frac{e^{\mathrm{i}z} - e^{-\mathrm{i}z}}{2\mathrm{i}}, \quad \cos(z) = \frac{e^{\mathrm{i}z} + e^{-\mathrm{i}z}}{2}.$$

`Sin(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field, return the power series $\sin(f)$.

`Sin(c): FldComElt -> FldComElt`

`Sin(r): FldReElt -> FldReElt`

Given a real or complex number c, return the value $\sin(c)$.

`Cos(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field, return the power series $\cos(f)$.

`Cos(c): FldComElt -> FldComElt`

`Cos(r): FldReElt -> FldReElt`

Given a real or complex number c, return the value $\cos(c)$.

`Sincos(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field, return the two power series $\sin(f)$ and $\cos(f)$.

`Sincos(s): FldReElt -> FldReElt, FldReElt`

`Sincos(s): FldComElt -> FldComElt, FldComElt`

Given a real or complex number s, return the two values $\sin(s)$ and $\cos(s)$.

`Tan(f): RngSerElt -> RngSerElt`

Given a power series f defined over the real or complex field, return the power series $\tan(f)$.

`Tan(c): FldComElt -> FldComElt`

`Tan(r): FldReElt -> FldReElt`

Given a real or complex number c, return the value $\tan(c) = \frac{\sin(c)}{\cos(c)}$. Note that c should not be too close to one of the zeroes $(\pi/2 + n \cdot \pi)$ of $\cos(z)$.

`Cot(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field having valuation zero, return the power series $\cot(f)$.

`Cot(c): FldComElt -> FldComElt`

`Cot(r): FldReElt -> FldReElt`

Given a real or complex number c, return the value $\cot(c) = \cos(c)/\sin(c)$. Note that c should not be too close to one of the zeroes $n \cdot \pi$ of $\sin(z)$.

`Sec(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field, return the power series $\sec(f)$.

`Sec(c): FldComElt -> FldComElt`

`Sec(r): FldReElt -> FldReElt`

Given a real or complex number c, return the value $\sec(c) = 1/\cos(c)$. Note that c should not be too close to one of the zeroes $(\pi/2 + n \cdot \pi)$ of $\cos(z)$.

`Cosec(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field having valuation zero, return the power series $\mathrm{cosec}(f)$.

`Cosec(c): FldComElt -> FldComElt`

`Cosec(r): FldReElt -> FldReElt`

Given a real or complex number c, return the value $\mathrm{cosec}(c) = 1/\sin(c)$. Note that c should not be too close to one of the zeroes $n \cdot \pi$ of $\sin(z)$.

26.5.3 Inverse Trigonometric Functions

The inverse trigonometric functions are all available for arbitrary real or complex arguments. The principal values are chosen as indicated. We mention the power series expansions for the inverse of the sine and tangent functions (for $|z| \leq 1$):

$$\arcsin(z) = \sum_{n=0}^{\infty} \left(\prod_{k=1}^{2n} k^{(-1)^{k-1}}\right) \frac{z^{2n+1}}{2n+1},$$

$$\arctan(z) = \sum_{n=1}^{\infty} (-1)^n \frac{z^{2n+1}}{2n+1}.$$

The important relations with the logarithmic function include

$$\arcsin(z) = \frac{1}{\mathrm{i}} \log(\mathrm{i}z + \sqrt{1 - z^2}),$$
$$\arccos(z) = \frac{1}{\mathrm{i}} \log(z + \sqrt{z^2 - 1}),$$
$$\arctan(z) = \frac{1}{2\mathrm{i}} \log(\frac{1 + \mathrm{i}z}{1 - \mathrm{i}z}).$$

`Arcsin(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field. return the inverse sine of the power series f.

`Arcsin(r): FldReElt -> FldReElt`

`Arcsin(r): FldComElt -> FldComElt`

Given a real or complex number s, return a value t such that $\sin(t) = s$. The principal value with real part in $[-\pi/2, \pi/2]$ is chosen. The return value is a complex number, unless s is real and $-1 \leq s \leq 1$, in which case a free real number is returned.

`Arccos(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field. return the inverse cosine of the power series f.

`Arccos(r): FldReElt -> FldReElt`

`Arccos(r): FldComElt -> FldComElt`

Given a real or complex number s, return a value t such that $\cos(t) = s$. The principal value with real part in $[0, \pi]$ is chosen. The return value is a complex number, unless s is real and $-1 \leq s \leq 1$, in which case a free real number is returned.

`Arctan(f): RngSerElt -> RngSerElt`

Given a power series f defined over the real or complex field, return the inverse tangent of the power series f.

`Arctan(r): FldReElt -> FldReElt`

`Arctan(r): FldComElt -> FldComElt`

Given a real or complex number s, return a value t such that $\tan(t) = s$. The principal value with real part in $(-\pi/2, \pi/2)$ is chosen. The return value is a complex number, unless s is real, in which case a free real number is returned.

`Arctan(x, y): FldReElt, FldReElt -> FldReElt`

`Arctan2(x, y): FldReElt, FldReElt -> FldReElt`

Given the real numbers x and y, return the value v of $\arctan(y/x)$ determined by the choice of signs for x and y. That is, the value v is chosen in $(-\pi, \pi)$ in such a way that the signs of y and $\sin(v)$ coincide, as well as the signs of x and $\cos(v)$. An error occurs if x and y are both zero; if y is zero and x non-zero, the value returned is $\mathrm{sign}(x) \cdot \pi/2$.

The arguments are allowed to be in any real field (automatic coercion is used whenever necessary). Note that the ordering of x and y differs from the C math library function.

`Arccot(r): FldReElt -> FldReElt`

`Arccot(r): FldComElt -> FldComElt`

Given a real or complex number s, return a value t such that $\cot(t) = s$. The principal value with real part in $(-\pi/2, \pi/2)$ is chosen. The return value is a complex number, unless s is real, in which case a real number is returned.

`Arcsec(r): FldReElt -> FldReElt`

`Arcsec(r): FldComElt -> FldComElt`

Given a real or complex number s, return a value t such that $\sec(t) = s$. The principal value with real part in $[0, \pi/2) \cup (\pi/2, \pi]$ is chosen. The return value is a complex number, unless s is real, in which case a real number is returned.

`Arccosec(r): FldReElt -> FldReElt`

`Arccosec(r): FldComElt -> FldComElt`

Given a real or complex number s, return a value t such that $\mathrm{cosec}(t) = s$. The principal value with real part in $[-\pi/2, 0) \cup (0, \pi/2]$ is chosen. The return value is a complex number, unless s is real, in which case a real number is returned.

26.5.4 Hyperbolic Functions

The hyperbolic functions are available for real and complex arguments, as specified below. The hyperbolic functions are defined using

$$\sinh(z) = \frac{e^z - e^{-z}}{2},$$
$$\cosh(z) = \frac{e^z + e^{-z}}{2}.$$

`Sinh(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field, return the hyperbolic sine of the power series f.

`Sinh(s): FldComElt -> FldComElt`

`Sinh(r): FldReElt -> FldReElt`

Given a real or complex number s, return $\sinh(s)$. The result will be a real or complex value, in accordance with the argument.

`Cosh(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field, return the hyperbolic cosine of the power series f.

`Cosh(r): FldReElt -> FldReElt`

`Cosh(r): FldComElt -> FldComElt`

Given a real or complex number s, return $\cosh(s)$. The result will be a real or complex value, in accordance with the argument.

`Tanh(f): RngSerElt -> RngSerElt`

Given a power series f defined over a real or complex field, return the hyperbolic tangent of the power series f.

`Tanh(r): FldReElt -> FldReElt`

`Tanh(r): FldComElt -> FldComElt`

Given a real or complex number s, return $\tanh(s) = \frac{\sinh(s)}{\cosh(s)}$. The result will be a real or complex value, in accordance with the argument.

`Coth(r): FldReElt -> FldReElt`

`Coth(r): FldComElt -> FldComElt`

Given a real or complex number s, return $\coth(s) = \frac{\cosh(s)}{\sinh(s)}$. The result will be a real or complex value, in accordance with the argument.

`Sech(r): FldReElt -> FldReElt`

`Sech(r): FldComElt -> FldComElt`

Given a real or complex number s, return $\mathrm{sech}(s) = 1/\cosh(s)$. The result will be a real or complex value, in accordance with the argument.

`Cosech(r): FldReElt -> FldReElt`

`Cosech(r): FldComElt -> FldComElt`

Given a real or complex number s, return $\mathrm{cosech}(s) = 1/\sinh(s)$. The result will be a real or complex value, in accordance with the argument.

26.5.5 Inverse Hyperbolic Functions

The inverse hyperbolic functions are available for real or complex arguments. The principal values are chosen as indicated.

Argsinh(f): RngSerElt -> RngSerElt

Given a power series f defined over a real or complex field, return the inverse hyperbolic sine of the power series f.

Argsinh(r): FldReElt -> FldReElt

Argsinh(r): FldComElt -> FldComElt

Given a real or complex number s, return t such that $\sinh(t) = s$; the principal value with imaginary part in $[\pi/2, \pi/2]$ is chosen. The return value is a complex number, unless the argument is real, in which case a real number is returned.

Argcosh(f): RngSerElt -> RngSerElt

Given a power series f defined over a real or complex field, return the inverse hyperbolic cosine of the power series f.

Argcosh(r): FldReElt -> FldReElt

Argcosh(r): FldComElt -> FldComElt

Given a real or complex number s, return t such that $\cosh(t)) = s$; the principal value with imaginary part in $[0, \pi]$ is chosen. The return value is a complex number, unless the argument is real and $s \geq 1$, in which case a real number is returned.

Argtanh(f): RngSerElt -> RngSerElt

Given a power series f defined over a real or complex field, return the inverse hyperbolic tangent of the power series f.

Argtanh(s): FldReElt -> FldReElt

Argtanh(r): FldComElt -> FldComElt

Given a real or complex number s, return t such that $\tanh(t) = s$; the principal value with imaginary part in $[\pi/2, \pi/2]$ is chosen. The return value is a complex number, unless the argument is real and $-1 < s < 1$, in which case a real number is returned.

Argsech(s): FldReElt -> FldReElt

Argsech(r): FldComElt -> FldComElt

Given a real or complex number s, return t such that $\text{sech}(t)) = s$; the principal value with imaginary part in $[0, \pi]$ is chosen. The return value is a complex number, unless the argument is real and $|s| \geq 1$, in which case a real number is returned.

Argcosech(s): FldReElt -> FldReElt

Argcosech(r): FldComElt -> FldComElt

Given a real or complex number s, return t such that $\text{cosech}(t)) = s$; the principal value with imaginary part in $[-\pi/2, \pi/2]$ is chosen. The return value is a complex number, unless the argument is real, in which case a real number is returned.

Argcoth(s): FldReElt -> FldReElt

Argcoth(r): FldComElt -> FldComElt

Given a real or complex number s, return t such that $\coth(t)) = s$; the principal value with imaginary part in $[-\pi/2, \pi/2]$ is chosen. The return value is a complex number, unless the argument is real and $0 < s \leq 1$, in which case free real number is returned.

26.6 Elliptic and Modular Functions

More information on elliptic functions can be found for example in Chandrasekharan [Cha85], and for modular functions and their use see Koblitz [Kob84].

26.6.1 Eisenstein Series

Let $f(z)$ be a modular function. Then $f(z)$ may be written as a Fourier series $f(z) = \sum_{n\in\mathbb{Z}} a_n q^n$, where $q = e^{2\pi iz}$, which has at most finitely many nonzero coefficients a_n with $n < 0$. Such a Fourier expansion of a modular function is called its *q-expansion*. In this and the next section we present intrinsics for q-expansions of the Eisenstein series and the Weierstrass $\wp$-function.

Let z be a point in the upper half-plane and let L be a lattice in $\mathbb{C}$. The Eisenstein series are defined as the coefficients of the Laurent Series expansion of the Weierstrass $\wp$-function:

$$\wp(z, L) = \frac{1}{z^2} + \sum_{2\leq k} G_k(L)(2k-1)z^{2k-2}$$

where $G_k(L)$ are the Eisenstein series. The normalization $E_{2n}(z) = \frac{1}{2\zeta(2n)} G_{2n}(z)$ ensures that $E_{2n}(z)$ has a *rational* q-expansion.

`Eisenstein(k, z): RngIntElt, RngSerElt -> RngSerElt`

Precision: RngIntElt	Default:

Given a positive even integer $k = 2n$ and a complex power series z with positive valuation, return the q-expansion of the normalized Eisenstein series $E_{2n}(z)$. If z has finite precision this is the default for `Precision` otherwise the default precision of the parent of z is used.

`Eisenstein(k, t): RngIntElt, FldComElt -> FldComElt`

Given a positive even integer $k = 2n$ and a point t in the upper half plane, return the value of $E_{2n}(z)$ at t.

`Eisenstein(k, L): RngIntElt, SeqEnum -> FldComElt`

Given a positive even integer $k = 2n$ and a lattice $L = [a, b]$ in the complex plane, return the value of the Eisenstein series $E_{2n}(z)$ relative to the lattice L.

`Eisenstein(k, F): RngIntElt, QuadBinElt -> RngSerElt`

Given a positive even integer $k = 2n$ and a binary quadratic form $F = ax^2 + bxy + cy^2$, return the value of the Eisenstein series $E_{2n}(z)$ at the point $\tau = \left(-b + \sqrt{b^2 - 4ac}\right)/(2a)$ where z is $e^{2\pi * \mathrm{i} * \tau}$.

Example: q-Expansion of an Eisenstein Series `(ex-db3e10)`

We compute the q-expansion for the normalized Eisenstein series $E_4(z)$.

```
> C<i> := ComplexField();
> R<z> := PowerSeriesRing(C);
> E4<q> := Eisenstein(4, z);
> E4;
1.0000000000000000000000000000 +
    240.0000000000000000000000000000*q +
    2160.0000000000000000000000000000*q^2 +
    6720.0000000000000000000000000000*q^3 +
    17520.0000000000000000000000000000*q^4 +
    30240.0000000000000000000000000000*q^5 +
    60480.0000000000000000000000000000*q^6 +
    82560.0000000000000000000000000000*q^7 +
    140400.0000000000000000000000000000*q^8 +
    181680.0000000000000000000000000000*q^9 +
    272160.0000000000000000000000000000*q^10 +
    319680.0000000000000000000000000000*q^11 +
    490560.0000000000000000000000000000*q^12 +
    527520.0000000000000000000000000000*q^13 +
    743040.0000000000000000000000000000*q^14 +
    846720.0000000000000000000000000000*q^15 +
    1123440.0000000000000000000000000000*q^16 +
```

(continues on next page)

(continued from previous page)

```
    1179360.00000000000000000000000*q^17 +
    1635120.00000000000000000000000*q^18 +
    1646400.00000000000000000000000*q^19 + O(q^20)
```

We now evaluate this series at the point $z1 = 2.5 + i$. Since the expansion is in terms of q rather than z we first must calculate the point $q1$ that corresponds to $z1$.

```
> q1 := Exp(2*Pi(RealField())*i*(2.5 +i));
> Evaluate(E4, q1);
0.55930285285619077373766762411942 +
3.67329046709782088758389413820E-31*i
```

If we are interested only in the value of E_4 at a single point, then we can compute it directly:

```
> Eisenstein(4, 2.5 + i);
0.55930285285619077373766762411942 +
3.67329046709782088758389413820E-31*i
```

26.6.2 Weierstrass Series

`WeierstrassSeries(z, q): RngSerElt, RngSerElt -> RngSerElt`

```
Precision: RngIntElt                    Default:
```

Return a normalized q-expansion of the Weierstrass $\wp$-function:

$$\wp(z, L) = \frac{1}{z^2} + \sum_{2 \le k} G_k(L)(2k-1)z^{2k-2}$$

where $G_k(L)$ are the Eisenstein series and

$$\texttt{WeierstrassSeries}(z, q) = (2\pi i)^{-2}\wp(q, z/(2\pi i))$$

Each term is an Eisenstein series, calculated to precision `Precision`, which is by default the precision of q.

`WeierstrassSeries(z, t): RngSerElt, FldComElt -> RngSerElt`

Given a complex power series z with positive valuation and a point $t = \tau$ in the upper-half complex plane, return the normalized q-expansion of the Weierstrass $\wp$-function. This is equivalent to evaluating the q-series expansion at $q = e^{2\pi i\tau}$.

`WeierstrassSeries(z, L): RngSerElt, SeqEnum -> RngSerElt`

Given a complex power series z with positive valuation and a lattice $L = [a, b]$ in the complex plane, returns the normalized q-expansion of the Weierstrass $\wp$-function relative to the lattice L.

`WeierstrassSeries(z, F): RngSerElt, QuadBinElt -> RngSerElt`

Given a complex power series z with positive valuation and a binary quadratic form $F = ax^2 + bxy + cy^2$, this function returns the q-expansion of the Weierstrass $\wp$-function at $\tau = \left(-b + \sqrt{b^2 - 4ac}\right)/(2a)$.

26.6.3 The Jacobi θ and Dedekind η-functions

The first Jacobi θ-function, $\theta(q, z)$, is defined by

$$\theta(q, z) = \frac{1}{\mathrm{i}} \sum_{n=-\infty}^{\infty} (-1)^n q^{(n+\frac{1}{2})^2} e^{(2n+1)\mathrm{i}z} = 2\sum_{n=0}^{\infty} (-1)^n q^{(n+\frac{1}{2})^2} \sin(2n+1)z.$$

Defined this way, θ satisfies $\theta(q, -z) = -\theta(q, z)$, it is periodic with period 2π in the second variable: $\theta(q, z + 2\pi) = \theta(q, z)$, and its zeroes are of the form $m_1\pi + m_2\frac{\log x}{\mathrm{i}}$ for any integers m_1, m_2.

`JacobiTheta(q, z): FldReElt, RngSerElt[FldRe] -> RngSerElt`

`JacobiTheta(q, z): FldComElt, RngSerElt[FldCom] -> RngSerElt`

For a real or complex number q satisfying $|q| < 1$, return the first of Jacobi's theta functions $\theta(q, z)$ as a power series expansion in z, a series over the complex numbers. Pari is used here.

`JacobiTheta(q, z): FldReElt, FldReElt -> FldReElt`

`JacobiTheta(q, z): FldComElt, FldComElt -> FldComElt`

For real or complex numbers q, z satisfying $|q| < 1$, return the value of $\theta(q, z)$, the first of Jacobi's theta functions. Pari is used here.

`JacobiThetaNullK(q, k): FldReElt, RngIntElt -> FldReElt`

For integer $k \geq 0$, return the k-th derivative $\theta^{(k)}(q, 0)$ of $\theta(q, z)$ at $z = 0$. Pari is used here.

`DedekindEta(z): RngSerElt -> RngSerElt`

Given a complex power series z with positive valuation, return the q-expansion of Dedekind's η-function. Note that the unnormalized series is returned, that is, the factor $q^{1/24}$ is *not* removed. See [Lan87].

`DedekindEta(s): FldComElt -> FldComElt`

For complex argument s with positive imaginary part, this returns the actual value of Dedekind's η-function which is defined by $\eta(s) = e^{\frac{2\pi i s}{24}} \left(1 + \sum_{n=1}^{\infty} (-1)^n (q^{n(3n-1)/2} + q^{n(3n+1)/2})\right)$ where $q = e^{2\pi i s}$.

26.6.4 The j-Invariant and the Discriminant

The discriminant of the elliptic curve corresponding to the complex lattice L_τ, spanned by 1 and τ is given by

$$\Delta(\tau) = q \left(1 + \sum_{n=1}^{\infty} (-1)^n (q^{n(3n-1)/2} + q^{n(3n+1)/2})\right)$$

where $q = e^{2\pi i \tau}$.

`jInvariant(q): RngSerElt -> RngSerElt`

Given a power series q over a real or complex field with positive valuation, return the q-expansion of the elliptic j-invariant. The expansion begins with

$$j(q) = q^{-1} + 744 + 196884q + \cdots .$$

Note that:

$$j(q) = \frac{E_4(q)^3}{\Delta(q)}$$

where $E_4(q) =$ `Eisenstein(4, q)` and $\Delta(q) =$ `Delta(q)`.

`jInvariant(s): FldComElt -> FldComElt`

For complex argument s with positive imaginary part, this returns the value of the elliptic j-invariant at s. This is a modular function of weight 0 whose Fourier expansion starts with

$$j(s) = e^{-2\pi i s} + 744 + 196884 e^{2\pi i s} + \cdots .$$

`jInvariant(L): SeqEnum -> FldComElt`

Given a lattice $L = [a, b]$ in the complex plane, this function returns the value of the elliptic j-invariant of L. This is the j-invariant of τ where $\tau = a/b$ or $\tau = b/a$, whichever is in the upper half complex plane.

`jInvariant(F): QuadBinElt -> FldComElt`

For a binary quadratic form $F = ax^2 + bxy + cy^2$ with negative discriminant, this returns the elliptic j-invariant of F. This is the j-invariant of τ where $\tau = \left(-b + \sqrt{b^2 - 4ac}\right)/(2a)$.

`Delta(z): RngSerElt -> RngSerElt`

Given a complex power series z, this function returns a q-series expansion of the discriminant $\Delta(z)$.

`Delta(t): FldComElt -> FldComElt`

Given a point t in the upper half plane, return the q-series expansion of the discriminant $\Delta(q)$ evaluated at $q = e^{2\pi i t}$.

`Delta(L): SeqEnum -> FldComElt`

Given a pair $L = [a,b]$ of complex numbers generating a lattice in $\mathbb{C}$, return the q-series expansion of the discriminant $\Delta(q)$ evaluated at $q = e^{2\pi i \tau}$ where $\tau = a/b$ or $\tau = b/a$, whichever is in the upper half complex plane.

26.6.5 Weber's Functions

`WeberF(s): FldComElt -> FldComElt`

For complex argument s in the upper half-plane, this returns the value of Weber's function f, defined in such a way that

$$j(s) = \frac{(f(s)^{24} - 16)^3}{f(s)^{24}}.$$

`WeberF2(g): RngSerElt -> RngSerElt`

For a complex power series g having positive valuation, this function returns the q-expansion of Weber's f_2 function

$$f_2(x) = \frac{\eta(2x)\sqrt{2}}{\eta(x)}$$

defined in such a way that

$$j(s) = \frac{(f_2(s)^{24} + 16)^3}{f_2(s)^{24}}.$$

`WeberF1(s): FldComElt -> FldComElt`

`WeberF2(s): FldComElt -> FldComElt`

For complex number s lying in the upper half-plane, these return the value of Weber's functions f_1 and f_2, defined in such a way that

$$j(s) = \frac{(f_{1/2}(s)^{24} + 16)^3}{f_{1/2}(s)^{24}}.$$

In fact, f_2 is as defined above and

$$f_1(x) = f_2(-1/x) = \eta(x/2)/\eta(x).$$

Example: q-Expansion of a Weber Function (ex-3718fd)

We compute the q-expansion for the Weber function $f_2(z)$.

```
> C<i> := ComplexField();
> R<x> := PowerSeriesRing(C);
> f2<q> := WeberF2(x);
> f2;
1.41421356237309504880168872421 +
    (1.41421356237309504880168872421 +
    0.37024024484653052058465674917 2*i)*q +
    (1.36574922765338060759226121771 +
```

(continues on next page)

(continued from previous page)

```
0.37024024484653052058465674917 2*i)*q^2 +
(2.77996279002647565639394994192 +
0.36601093379329241948227297708 1*i)*q^3 +
(2.78023959778761313408864734217 +
0.73625117863982294006692972625 3*i)*q^4 +
(4.14598882544099374168090855987 +
0.73626567226030370903683781952 8*i)*q^5 +
(5.56020175541059398072755542234 +
1.10227660605359612851911079661*i)*q^6 +
...
```

26.7 Theta Functions

One of the main tools for working with analytic Jacobians is the theta function. For instance it is used by `FromAnalyticJacobian(z, A): Mtrx, AnHcJac → SeqEnum` and `RosenhainInvariants(t): Mtrx → Set`. For $c \in \mathbb{R}^{2g}$ let c' be the first g entries and c'' the second g entries of c. For such a c, $z \in \mathbb{C}^g$ and τ an element of Siegel upper half-space the classical multi-variable theta function is defined by

$$\theta[c](z,\tau) = \sum_{m \in \mathbb{Z}^g} \exp(\pi i^t (m+c')\tau(m+c') + 2\pi i^t (m+c')(z+c'')).$$

The vector c is called the characteristic of the theta function.

`Theta(char, z, tau): Mtrx, Mtrx, Mtrx -> FldComElt`

This computes the multidimensional theta function with characteristic *char* (a $2g \times 1$ matrix) at z (a $g \times 1$ matrix) and τ (a symmetric $g \times g$ matrix with positive definite imaginary part).

`Theta(char, z, A): Mtrx, Mtrx, AnHcJac -> FldComElt`

This computes the multidimensional theta function with characteristic *char* (a $2g \times 1$ matrix) at z (a $g \times 1$ matrix) and τ, the small period matrix of the analytic Jacobian A. This function caches the values of theta null values ($z = 0$) at half-integer characteristics.

26.8 Gamma, Bessel, and Associated Functions

As a general reference to the functions described in this section (and much more), we refer the reader to Whittaker and Watson [WW15].

`Gamma(f): RngSerElt -> RngSerElt`

Return the Gamma function $\Gamma(f)$ of the series f. The series f must be defined over the free real or complex field, the valuation of f must be 0 and the constant term of f must be 1.

`Gamma(r): FldReElt -> FldReElt`

`Gamma(r): FldComElt -> FldComElt`

Given a real or complex number s (not equal to $0, -1, -2, \ldots$), calculate the value $\Gamma(s)$ of the *gamma function* at s. For s with positive real part this is the value of

$$\Gamma(s) = \int_0^\infty u^{s-1} e^{-u} du.$$

For other s (not a non-positive integer) the function is defined by analytic continuation, and it satisfies the product formula

$$\frac{1}{s\Gamma(s)} = e^{\gamma s} \prod_{n=1}^{\infty} (1 + \frac{s}{n}) e^{-s/n}.$$

The function Γ also satisfies

$$\Gamma(s)\Gamma(1-s) = \frac{\pi}{\sin(\pi s)},$$

and

$$\Gamma(s+1) = s\Gamma(s).$$

`Gamma(s, t): FldReElt, FldReElt -> FldReElt`

`Complementary:`	`BoolElt`	`Default: false`
`Gamma :`	`FldReElt`	`Default:`

For real numbers s, t this returns the value of the incomplete gamma function

$$\gamma(s,t) = \int_0^t u^{s-1}e^{-u}du.$$

The optional argument `Complementary` can be used to find the complement

$$\int_t^\infty u^{s-1}e^{-u}du$$

instead. There is a second optional argument that may be used in the computation of the incomplete gamma value; the free real value of `Gamma` should be the value of $\Gamma(s)$, in which case $\gamma(s,t)$ may be computed as the difference between the given value for $\Gamma(s)$ and that of the complementary γ at s, t. Pari is used here.

`GammaD(s): FldReElt -> FldReElt`

For free real s (such that $s+\frac{1}{2}$ is not a non-positive integer) this returns the value of $\Gamma(s+\frac{1}{2})$. For integer values of s this is faster than `Gamma(s+(1/2))`, because Legendre's doubling formula

$$\Gamma(s+\frac{1}{2}) = 2^{1-2s}\sqrt{\pi}\frac{\Gamma(2s)}{\Gamma(s)}$$

is used. Pari is used here.

`LogGamma(f): RngSerElt -> RngSerElt`

Return the Log-Gamma function $\mathrm{Log}(\Gamma(f))$ of the series f. f must be defined over a real or complex field, the valuation of f must be 0 and the constant term of f must be 1.

`LogGamma(r): FldReElt -> FldReElt`

`LogGamma(r): FldComElt -> FldComElt`

For real or complex s (not a non-positive integer) return the value of the principal branch of the logarithm of the gamma function of s.

`LogDerivative(s): FldReElt -> FldReElt`

`LogDerivative(s): FldComElt -> FldComElt`

`Psi(s): FldReElt -> FldReElt`

`Psi(s): FldComElt -> FldComElt`

For real or complex s (not a non-positive integer) return the principal value of the logarithmic derivative

$$\Psi(s) = \frac{d\log\Gamma(s)}{ds} = \frac{\Gamma'(s)}{\Gamma(s)},$$

of the gamma function, which allows the expansion

$$\Psi(s) = -\gamma - \frac{1}{s} + s\sum_{n=1}^{\infty}\frac{1}{n(s+n)};$$

here γ is Euler's gamma. Pari is used here.

`BesselFunction(n, r): RngIntElt, FldReElt -> FldReElt`

Given a small integer n and a real number r, calculate the value of the *Bessel function* $y = J_n(r)$, of the first kind of order n. Results for negative arguments are defined by: $J_{-n}(r) = J_n(-r) = (-1)^n J_n(r)$. The Bessel function of the first kind of order n is defined by

$$J_n(x) = \frac{1}{2\pi \mathrm{i}} \left(\frac{z}{2}\right)^n \int_{-\infty}^{0^+} u^{-n-1} e^{u - \frac{z^2}{4t}} du,$$

and satisfies

$$J_n(x) = \sum_{k=0}^{\infty} \frac{(-1)^k z^{n+2k}}{2^{n+2k} k! \Gamma(n+k+1)}.$$

`BesselFunctionSecondKind(n, r): RngIntElt, FldReElt -> FldReElt`

Given a small integer n and a real number r, calculate the value of the *Bessel function* $y = Y_n(r)$, of the second kind of order n. Results for negative arguments are defined by: $Y_{-n}(r) = -(-1)^n Y_n(r)$, $Y_n(-r)$ is not a real number. The Bessel function of the second kind of order n satisfies the Bessel differential equation.

`JBessel(n, s): RngIntElt, FldReElt -> FldReElt`

`JBessel(n, s): FldReElt, FldReElt -> FldReElt`

Given a small integer n and a real number s, calculate the value of the *Bessel function* of the first kind of half integral index $n + \frac{1}{2}$, $J_{n+\frac{1}{2}}$, defined as above. Pari is used here.

`KBessel(n, s): FldReElt, FldReElt -> FldReElt`

`KBessel(n, s): FldComElt, FldReElt -> FldComElt`

`KBessel2(n, s): FldReElt, FldReElt -> FldReElt`

`KBessel2(n, s): FldComElt, FldReElt -> FldComElt`

Given a complex n and a positive real s, compute the value of the *modified Bessel function of the second kind* $K_n(s)$, which may be defined by

$$K_n(s) = \frac{\pi}{2} \left(\mathrm{i}^n J_{-n}(\mathrm{i}s) - \mathrm{i}^{-n} J_n(s)\right) \cot(n\pi).$$

The function `KBessel2` is an alternative (often faster) implementation of this function. Pari is used here.

26.9 The Hypergeometric Function

For more information on the Hypergeometric Series, see Husemöller [Hus87], page 176.

`HypergeometricSeries(a, b, c, z): RngElt, RngElt, RngElt, RngElt -> RngElt`

Return the hypergeometric series $F(a, b, c; z)$ defined by

$$F(a, b, c; z) = \sum_{0 \le n} \frac{(a)_n (b)_n}{n! (c)_n z^n}$$

where $(a)_n = a(a+1)\cdots(a+n-1)$.

`HypergeometricU(a, b, s): FldReElt, FldReElt, FldReElt -> FldReElt`

For positive real s and complex arguments a and b this function returns the value of the confluent hypergeometric function $U(a, b, s)$. This can be defined by

$$U(a, b, s) = \frac{1}{\Gamma(a)} \int_{u=0}^{\infty} e^{-su} u^{a-1} (1+u)^{b-a-1)} du.$$

Pari is used here.

26.10 Other Special Functions

`ArithmeticGeometricMean(x, y): RngSerElt, RngSerElt -> RngSerElt`

`AGM(f, g): RngSerElt, RngSerElt -> RngSerElt`

Return the hyperbolic arithmetic-geometric mean of the series f and g defined over a field. The valuations of f and g must be equal.

`ArithmeticGeometricMean(x, y): FldReElt, FldReElt -> FldReElt`

`AGM(x, y): FldReElt, FldReElt -> FldReElt`

Returns the arithmetic-geometric mean of the real or complex numbers x and y, defined as the limit of either of the sequences x_i, y_i where $x_0 = x$, $y_0 = y$, and $x_{i+1} = (x_i + y_i)/2, y_{i+1} = \sqrt{x_i y_i}$. The function calculates both sequences, and when the numbers are within the desired precision of each other, it returns one of them.

`BernoulliNumber(n): RngIntElt -> FldRatElt`

For a non-negative integer n, return the value of the n-th Bernoulli number B_n, defined by

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!}.$$

`BernoulliApproximation(n): RngIntElt -> FldReElt`

For a non-negative integer n, return an approximation in the field of real numbers to the value of the n-th Bernoulli number B_n, defined by

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!}.$$

`DawsonIntegral(r): FldReElt -> FldReElt`

Given a real number r, compute the value of *Dawson's integral*,

$$e^{-x^2} \cdot \int_0^x e^{u^2} du,$$

at $x = r$.

`ErrorFunction(r): FldReElt -> FldReElt`

`Erf(r): FldReElt -> FldReElt`

Given a real number r, calculate the value of the *error function* erf. This is the value of

$$\sqrt{\frac{4}{\pi}} \cdot \int_0^x e^{-u^2} du,$$

at $x = r$ for $r > 0$, and for $r < 0$ it is defined by $\mathrm{erf}(x) = -\mathrm{erf}(-x)$, while $\mathrm{erf}(0) = 0$.

`ComplementaryErrorFunction(r): FldReElt -> FldReElt`

`Erfc(r): FldReElt -> FldReElt`

Given a real number r, calculate the value of the *complementary error function*. This is the value of $y = \mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$ for the error function erf as defined above.

`ExponentialIntegral(r): FldReElt -> FldReElt`

Given a real number r, calculate the value of the *exponential integral*, that is, the principal value of

$$\int_{-\infty}^{x} \frac{e^u}{u} du$$

at $x = r$.

`ExponentialIntegralE1(r): FldReElt -> FldReElt`

Given a real number r, calculate the value of the *exponential integral E1*, that is, the principal value of

$$\int_x^\infty \frac{e^{-u}}{u} du$$

at $x = r$.

`LogIntegral(r): FldReElt -> FldReElt`

Given a non-negative real number r that is not equal to 1, evaluate the *logarithmic integral* $y = \mathrm{li}(x)$ at $x = r$. This integral is defined to be the principal value of $\int_0^x \frac{1}{\log(u)} du$.

`ZetaFunction(s): FldReElt -> FldReElt`

`ZetaFunction(R, n): FldRe, RngIntElt -> FldReElt`

These functions calculate values of the Riemann ζ-function, which is the analytic continuation of

$$\zeta(z) = \sum_{i=1}^{\infty} \frac{1}{i^z}$$

(convergent for $\mathrm{Re}(z) > 1$). The version with one argument takes a real or complex number $r \neq 1$ and returns a real or complex number. The version with two arguments is much more restricted; it takes a real field R and an integer $n \neq 1$, and returns $\zeta(n)$ in R.

MPFR uses the algorithm of Jean-Luc Rémy and Sapphorain Pétermann [PR06].

26.11 Summation of Infinite Series

There are three functions for evaluating infinite sums of real numbers. The sum should be specified as a map m from the integers to the real field, such that $m(n)$ is the n^{th} term of the sum. The summation begins at term i. The precision of the result will be the default precision of the real field.

`InfiniteSum(m, i): Map, RngIntElt -> FldReElt`

An approximation to the infinite sum $m(i) + m(i+1) + m(i+2) + \cdots$. This function also works for maps to the complex field.

`PositiveSum(m, i): Map, RngIntElt -> FldReElt`

An approximation to the infinite sum $m(i) + m(i+1) + m(i+2) + \cdots$. Designed for series in which every term is positive, it uses van Wijngaarden's trick for converting the series into an alternating one. Due to the stopping criterion, terms equal to 0 will create problems and should be removed.

`AlternatingSum(m, i): Map, RngIntElt -> FldReElt`

```
Al: MonStgElt                    Default: "Villegas"
```

An approximation to the infinite sum $m(i) + m(i+1) + m(i+2) + \cdots$. Designed for series in which the terms alternate in sign. The optional argument `Al` can be used to specify the algorithm used. The possible values are `"Villegas"` (the default), and `"EulerVanWijngaarden"`. Due to the stopping criterion, terms equal to 0 will create problems and should be removed.

26.12 Numerical Integration

Two groups of intrinsics for numerical integration are provided. The first group are designed for the numerical integration of complex-valued functions along the interval $[-1, 1]$. They have been researched and implemented by Christian Neurohr. The second group consist of three basic Romberg-like numerical integration methods that are based on those in Pari.

26.12.1 Polynomial Interpolation

Given $n+1$ points (x_i, y_i), there is a unique polynomial $p(x)$ of degree n which passes through these points. Given some t, *polynomial interpolation* computes the value $u = p(t)$.

`Interpolation(P, V, t): [FldReElt], [FldReElt], FldReElt -> FldReElt, FldReElt`

Using Neville's algorithm, interpolate the value of t with respect to a polynomial p such that $p(P[i]) = V[i]$. An estimate of the error is also returned.

26.12.2 Discrete Fourier Transform

An intrinsic is provided to compute the discrete Fourier transform of a sequence of complex numbers.

`DiscreteFourierTransform(E): SeqEnum[FldComElt] -> SeqEnum[FldComElt]`

Given a sequence E of complex numbers, return their discrete Fourier transform as a sequence of complex numbers.

26.12.3 Integration of Complex Functions

In this section several integration schemes that are useful for numerical integration along the interval $[-1, 1]$ are described. They have been designed and implemented by Christian Neurohr. In particular, the routines are suitable for numerically approximating integrals of complex-valued functions that are analytic over the interval [-1,1] or have integrable singularities at the end points. These should not be regarded as general purpose numerical integration tools. The examples in this section merely indicate how to use these integration methods. Error bounds for them and more details on how to use them can be found in [Neu18, Chapter 3] as well as many other references on numerical analysis.

Gaussian Quadratures

Gaussian quadratures are well-known quadrature rules that integrate exactly polynomials of degree $2N-1$ using N integration points. In particular, they are highly efficient for integrands that are well-behaved around the integration path. The following functions compute the *Gauss-Jacobi integration scheme* on the interval $[-1, 1]$, i.e. a sequence of abscissas A and a sequence of weights W such that can be used to approximate the definite integral

$$\int_{-1}^{1} (1-x)^a (1+x)^b f(x) dx \approx \sum_{k=1}^{N} W[k] \cdot A[k]$$

for a well-behaved, complex-valued function $f : [-1, 1] \to \mathbb{C}$, $a, b > -1$ and N sufficiently large. The *Gauss-Legendre integration scheme* occurs as the special case $a = b = 0$. For more details see also [Neu18, Section 3.2].

`GaussLegendreIntegrationPoints(N, D): RngIntElt, RngIntElt -> SeqEnum[FldReElt], SeqEnum[FldReElt]`

Construct the Gauss–Legendre integration scheme on N points and precision D.

`GaussJacobiIntegrationPoints(N, D, a, b): RngIntElt, RngIntElt, RngReSubElt, RngReSubElt -> SeqEnum, SeqEnum`

Construct the Gauss–Jacobi integration scheme on N points and precision D.

Clenshaw–Curtis Quadrature

The Clenshaw–Curtis quadrature on $N+1$ is an interpolation integration scheme that integrates polynomials exactly up to degree N. The abscissas are the $N-1$ extrema of the N-th Chebyshev polynomial plus the boundary points ± 1 and the weights are obtained by interpolation. It is best suited for integrands that are analytic around $[-1, 1]$ and can be computed quite quickly using a discrete Fourier transform. As before, a sequence of abscissas A and a sequence of weights W on interval $[-1, 1]$ can be used to approximate

$$\int_{-1}^{1} f(x) dx \approx \sum_{k=1}^{N} W[k] \cdot A[k]$$

for a well-behaved, complex-valued function $f : [-1, 1] \to \mathbb{C}$ and N chosen large enough. For more details on Clenshaw–Curtis integration and this particular implementation see [Neu18, Section 3.3].

ClenshawCurtisIntegrationPoints(N, D): RngIntElt, RngIntElt -> SeqEnum[FldReElt], SeqEnum[FldReElt]

Construct the Clenshaw–Curtis integration scheme on $N + 1$ points and precision D.

Tanh–Sinh Quadrature

Double–exponential integration (or tanh–sinh quadrature) is a quite versatile integration scheme. While it is far less efficient than Gaussian or Clenshaw–Curtis quadrature for well-behaved integrands, double-exponential integration can easily handle difficult integrands with endpoint singularities. In fact, the double–exponential integration as implemented here can handle integrands similar to Gauss–Jacobi quadrature. It returns three sequences of real numbers of length $2N + 1$: abscissas A, weights W_1, and 'extra weights' W_2.

$$\int_{-1}^{1} (1-x)^a (1+x)^b f(x)dx \approx \sum_{k=1}^{2N+1} W_1[k] \cdot W_2[k]^b \cdot (1 + A[k])^{b-a} \cdot f(A[k]) \,.$$

for a well–behaved, complex–valued function $f : [-1, 1] \to \mathbb{C}$ and N chosen large enough and, without loss of generality, $b > a$.

TanhSinhIntegrationPoints(N, h): RngIntElt, FldReElt -> SeqEnum[FldReElt], SeqEnum[FldReElt], SeqEnum[FldReElt]

Construct the double–exponential integration scheme with $2N + 1$ points and step-length $h > 0$. The precision of the abscissas and weights is inherited from h.

Example: Numerical Integration on the Unit Interval (ex-fd7870)

The easy integral

$$\int_0^1 x \log(1+x)dx = \frac{1}{4} \,.$$

is chosen to illustrate how these integration schemes may be used. The computation is performed using precision $D = 50$.

```
> N := 20;
> D := 50;
> function f(x)
>    return x*Log(1+x);
> end function;
> A_GL, W_GL := GaussLegendreIntegrationPoints(N,D);
> /* Map abscissas to the interval [0,1] */
> A_GL := [ (1/2)*(a+1) : a in A_GL ];
> int_GL := (1/2) * &+[ W_GL[k] * f(A_GL[k]) : k in [1..N] ];
> Abs(int_GL - 1/4);
9.3232511485451973751923945899073760403381175208927E-33
```

Now, using Clenshaw–Curtis integration on $N + 1 = 21$ points

```
> A_CC, W_CC := ClenshawCurtisIntegrationPoints(N,D);
> A_CC := [ (1/2)*(a+1) : a in A_CC ];
> int_CC := (1/2) * &+[ W_CC[k] * f(A_CC[k]) : k in [1..N+1] ];
> Abs(int_CC - 1/4);
1.4644696752310341410357330113758869025438089669910E-20
```

and, finally, double-exponential with steplength $h = 6/N$

```
> h := RealField(D)!6/N;
> A_DE, W_DE := TanhSinhIntegrationPoints(N,h);
> A_DE := [ (1/2)*(a+1) : a in A_DE ];
> int_DE := (1/2) * &+[ W_DE[k] * f(A_DE[k]) : k in [1..2*N+1] ];
> Abs(int_DE - 1/4);
4.42234512943163562510023176911426688750959642809 41E-10
```

Example: Integral with Endpoint Singularities (ex-9327e9)

One could, for example, want to approximate the integral

$$\int_{-1}^{1} \frac{\cos(x)\sin(x)}{(1-x)^{\frac{1}{3}}(1+x)^{\frac{1}{3}}}dx \approx 0.12886634861848213601060873074 5\,.$$

using $N = 10$ integration points in precision $D = 30$.

```
> c := 0.12886634861848213601060873074 5;
> N := 10;
> D := 30;
> function f(x)
>    return Sin(x)*Cos(x);
> end function;
> A, W := GaussJacobiIntegrationPoints(N,D,-1/3,-1/5);
> int_GJ := &+[ W[k] * f(A[k]) : k in [1..N] ];
> Abs(int_GJ - c);
1.144062287304475972238367685 06E-20
```

A similar result can be achieved when using double-exponential integration (or Tanh–Sinh quadrature) with steplength $h = 1/8$ and $N = 48$

```
> h := Real(2^-3);
> N := Ceiling(6/h);
> A, W1, W2 := TanhSinhIntegrationPoints(N,h);
> int_DE := &+[ W1[k]*W2[k]^(1/3)*(1+A[k])^(1/3-1/5)*f(A[k]) : k in [1..2*N+1] ];
> Abs(int_DE - c);
1.32999219914892690177636985 694E-20
```

Example: Integral over an Infinite Interval (ex-1222c3)

The double-exponential integration can also be used to approximate tough integrals such as

$$\int_0^{\infty} e^{-x}\cos(x)dx = \int_0^1 \frac{e^{1-\frac{1}{z}}\cos(\frac{1}{z}-1)}{z^2}dz = \frac{1}{2}\,.$$

```
> h := RealField(1000)!(2^-10);
> N := Ceiling(7.2/h);
> A_DE, W_DE := TanhSinhIntegrationPoints(N,h);
> assert A_DE[1] gt -1;
> assert A_DE[2*N+1] lt 1;
> A_DE := [ (1/2)*(a+1) : a in A_DE ];
> function f(z)
>    return (Exp(1-1/z) * Cos(1/z-1)/z^2);
> end function;
> int_DE := (1/2) * &+[ W_DE[k] * f(A_DE[k]) : k in [1..2*N+1] ];
> Abs(int_DE - 1/2) lt 10^-300;
true
```

26.12.4 Romberg-Type Integration

A number of 'Romberg-Type' integration methods are provided which are based on Pari implementations. The precision should not be made too large for this, and the range of integration (including its boundaries) must be free of singularities.

RombergQuadrature(f, a, b: parameters): Program, FldReElt, FldReElt -> FldReElt

```
Precision: FldReElt                    Default: 1.0e-6
MaxSteps : RngIntElt                   Default: 20
K        : RngIntElt                   Default: 5
```

Using Romberg's method of order $2K$, approximate the integral of f from a to b. The desired accuracy may be specified by setting the `Precision` parameter, and the order of the algorithm by changing `K`. The algorithm ceases after `MaxSteps` iterations if the desired accuracy has not been achieved.

SimpsonQuadrature(f, a, b, n): Program, FldReElt, FldReElt, RngIntElt -> FldReElt

Using Simpson's rule on n sub-intervals, approximate the integral of f from a to b.

TrapezoidalQuadrature(f, a, b, n): Program, FldReElt, FldReElt, RngIntElt -> FldReElt

Using the trapezoidal rule on n sub-intervals, approximate the integral of f from a to b.

26.12.5 Numerical Derivatives

A function is provided to compute the numerical derivative of a function. This is achieved by computing enough interpolation points and performing a Taylor expansion.

NumericalDerivative(f, n, z): UserProgram, RngIntElt, FldComElt -> FldComElt

Given a suitably nice function f, compute a numerical approximation to the nth derivative at the point z.

Example: Numerical Derivative (ex-c82809)

```
> f := func<x|Exp(2*x)>;
> NumericalDerivative(f, 10, ComplexField(30)! 1.0) / f (1.0);
1024.000000000000000000000000
> NumericalDerivative(func<x|LogGamma(x)>,1,ComplexField()!3.0);
0.92278433509846713939348790918
> Psi(3.0); // Psi is Gamma'/Gamma
0.92278433509846713939348790918
```

26.13 Bibliography

- [Cha85] K. Chandrasekharan. *Elliptic Functions*. Volume 281 of Grundlehren der mathematischen Wissenschaften. Springer, Berlin, 1985.
- [Hus87] Dale Husemöller. *Elliptic Curves*. Volume 111 of Graduate Texts in Mathematics. Springer, New York, 1987.
- [KLL84] Ravi Kannan, Arjen K. Lenstra, and László Lovász. Polynomial Factorization and Nonrandomness of Bits of Algebraic and Some Transcendental Numbers. In *Proceedings of the 16th Symposium on the Theory of Computing (STOC 1984)*, 191–200. ACM, 1984.
- [Kob84] Neal Koblitz. *Introduction to Elliptic Curves and Modular Forms*. Volume 97 of Graduate Texts in Mathematics. Springer, New York, 1984.
- [Lan87] Serge Lang. *Elliptic Functions*. Volume 112 of Graduate Texts in Mathematics. Springer, New York, 1987.
- [LLL82] Arjen K. Lenstra, Hendrik W. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

- [Lew81] Leonard Lewin. *Polylogarithms and associated functions*. North Holland, New York, 1981.
- [Neu18] Christian Neurohr. *Efficient integration on Riemann surfaces & applications*. Dissertation, Carl von Ossietzky Universität Oldenburg, 2018. URL: http://oops.uni-oldenburg.de/3607.
- [PR06] Y.-F. S. Pétermann and Jean-Luc Rémy. Arbitrary Precision Error Analysis for computing $\zeta(s)$ with the Cohen-Olivier algorithm: Complete description of the real case and preliminary report on the general case. Research Report 5852, INRIA, 2006. http://www.inria.fr/rrrt/rr-5852.html.
- [Sch82] A. Schönhage. The fundamental theorem of algebra in terms of computational complexity. Technical Report, Univ. Tübingen, 1982.
- [vH01] Mark van Hoeij. Factoring Polynomials and 0-1 vectors. In *Proceedings of the Cryptography and Lattices Conference (CaLC 2001), Brown University, Providence, RI, USA, March 29-30, 2001*, 142–146. Springer, 2001.
- [vH02] Mark van Hoeij. Factoring Polynomials and the knapsack problem. *J. Number Th.*, 95(2):167–189, 2002. http://www.math.fsu.edu/∼hoeij/paper/knapsack.ps.
- [WW15] E. T. Whittaker and G. N. Watson. *A course of modern analysis*. Cambridge University Press, Cambridge, 2nd edition, 1915.
- [Zag] D. Zagier. "Polylogarithms, Dedekind Zeta Functions, and the Algebraic K-Theory of Fields." 377–390.

References

[AH99] L. M. Adleman and M.-D. A. Huang. Function field sieve method for discrete logarithms over finite fields. *Inform. and Comput.*, 151(1–2):5–16, 1999.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[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.

[AM93] A. O. L. Atkin and F. Morain. Elliptic curves and primality proving. *Math. Comp.*, 61:29 – 68, 1993.

[BM97] Laurent Bernardin and Michael B. Monagan. Efficient Multivariate Factorization Over Finite Fields. In *Proceedings of AAECC*, volume 1255 of LNCS, 15–28. Springer-Verlag, 1997.

[BCS97] Wieb Bosma, John Cannon, and Allan Steel. Lattices of Compatibly Embedded Finite Fields. *J. Symbolic Comp.*, 24(3):351–369, 1997.

[Bre80] R. P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20:176–184, 1980.

[BtR92] R. P. Brent and H. J. J te Riele. Factorizations of $a^n \pm 1$, $13 \leq a < 100$. Technical Report, Centrum voor Wiskunde en Informatica, Amsterdam, 1992. URL:ftp://nimbus.anu.edu.au/pub/Brent.

[Cha85] K. Chandrasekharan. *Elliptic Functions*. Volume 281 of Grundlehren der mathematischen Wissenschaften. Springer, Berlin, 1985.

[CGG89] Bruce W. Char, Keith O. Geddes, and Gaston H. Gonnet. GCDHEU: Heuristic Polynomial GCD Algorithm Based on Integer GCD Computation. *J. Symbolic Comp.*, 7(1):31–48, 1989.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Volume 138 of Graduate Texts in Mathematics. Springer, Berlin–Heidelberg–New York, 1993.

[Cop84] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Trans. Inform. Theory*, IT–30(4):587–594, July 1984.

[COS86] D. Coppersmith, A. M. Odlyzko, and R. Schroeppel. Discrete logarithms in GF(p). *Algorithmica*, 1:1–15, 1986.

[Cop96] Don Coppersmith. Finding a small root of a univariate modular equation. In *Advances in Cryptology—EuroCrypt 1996*, volume 1070 of LNCS, pages 155–165. Springer, 1996.

[Dem68] Peter Dembowski. *Finite geometries*. Ergebnisse der Mathematik und ihrer Grenzgebiete, Band 44. Springer-Verlag, Berlin, 1968.

[Dem71] Peter Dembowski. Generalized Hughes planes. *Canad. J. Math.*, 23:481–494, 1971.

[Dic05a] Leonard Eugene Dickson. Definitions of a group and a field by independent postulates. *Trans. Amer. Math. Soc.*, 6(2):198–204, 1905.

[Dic05b] Leonard Eugene Dickson. On finite algebras. *Nachr. Kgl. Ges. Wiss. Göttingen, Math.-phy. Klasse*, pages 358–393, 1905.

[EK63] Erich Ellers and Helmut Karzel. Kennzeichnung elliptischer Gruppenräume. *Abh. Math. Sem. Univ. Hamburg*, 26:55–77, 1963.

[EK64] Erich Ellers and Helmut Karzel. Endliche Inzidenzgruppen. *Abh. Math. Sem. Univ. Hamburg*, 27:250–264, 1964.

[FKMW04] J. Franke, T. Kleinjung, F. Morain, and T. Wirth. Proving the primality of very large numbers with fastECPP. In *Algorithmic number theory*, volume 3076 of Lecture Notes in Comput. Sci., pages 194–207. Springer, Berlin, 2004.

[GCL92] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Kluwer, Boston/Dordrecht/London, 1992.

[GM93] D. M. Gordon and K. S. McCurley. Massively parallel computation of discrete logarithms. In Ernest F. Brickell, editor, *Advances in Cryptology—CRYPTO 1992*, volume 740 of LNCS, 312–323. Springer-Verlag, 1993. Proc. 12th Annual International Cryptology Conference, Santa Barbara, Ca, USA, August 16–20, 1992.

[Gro74] Susan Dancs Groves. *Locally finite near-fields*. PhD thesis, Australian National University, 1974.

[Hal59] Marshall Hall, Jr. *The theory of groups*. The Macmillan Co., New York, N.Y., 1959.

[Hug57] D. R. Hughes. A class of non-Desarguesian projective planes. *Canad. J. Math.*, 9:378–388, 1957.

[Hus87] Dale Husemöller. *Elliptic Curves*. Volume 111 of Graduate Texts in Mathematics. Springer, New York, 1987.

[JL06] A. Joux and R. Lercier. The function field sieve in the medium prime case. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of Lecture Notes in Comput. Sci., 254–270. Berlin, 2006. Springer.

[KS95] Erich Kaltofen and Victor Shoup. Subquadratic-time factoring of polynomials over finite fields. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, 398–406. ACM, 1995.

[KLL84] Ravi Kannan, Arjen K. Lenstra, and László Lovász. Polynomial Factorization and Nonrandomness of Bits of Algebraic and Some Transcendental Numbers. In *Proceedings of the 16th Symposium on the Theory of Computing (STOC 1984)*, 191–200. ACM, 1984.

[Knu97] Donald E. Knuth. *The Art of Computer Programming*. Volume 2. Addison Wesley, Reading, Massachusetts, 3rd edition, 1997.

[Kob84] Neal Koblitz. *Introduction to Elliptic Curves and Modular Forms*. Volume 97 of Graduate Texts in Mathematics. Springer, New York, 1984.

[Lun69] Heinz Lüneburg. *Lectures on projective planes*. University of Illinois, Chicago, 1969.

[Lun71] Heinz Lüneburg. Über die Anzahl der Dickson'schen Fastkörper gegebener Ordnung. In *Atti del Convegno di Geometria Combinatoria e sue Applicazioni (Univ. Perugia, Perugia, 1970)*, pages 319–322. Ist. Mat., Univ. Perugia, Perugia, 1971.

[LO91a] B. A. LaMacchia and A. M. Odlyzko. Computation of Discrete Logarithms in Prime Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of INCS, 616–618. Springer-Verlag, 1991.

[LO91b] B. A. LaMacchia and A. M. Odlyzko. Solving Large Sparse Linear Systems over Finite Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of LNCS, 109–133. Springer-Verlag, 1991.

[Lan87] Serge Lang. *Elliptic Functions*. Volume 112 of Graduate Texts in Mathematics. Springer, New York, 1987.

[LD95] Arjen K. Lenstra and Bruce Dodson. NFS with four large primes: An explosive experiment. In Don Coppersmith, editor, *Advances in cryptology—CRYPTO 1995*, volume 963 of LNCS, 372–385. Berlin, 1995. Springer.

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

[Lew81] Leonard Lewin. *Polylogarithms and associated functions*. North Holland, New York, 1981.

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

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

[Mar92] George Marsaglia. The mathematics of random number generators. In *The unreasonable effectiveness of number theory (Orono, ME, 1991)*, volume 46 of Proc. Sympos. Appl. Math., pages 73–90. Amer. Math. Soc., Providence, RI, 1992. URL: http://dx.doi.org/10.1090/psapm/046/1195842.

[MZ91] George Marsaglia and Arif Zaman. A new class of random number generators. *Ann. Appl. Probab.*, 1(3):462–480, 1991.

[May03] Alexander May. *New RSA Vulnerabilities Using Lattice Reduction Methods*. Dissertation, University of Paderborn, 2003.

[Mon92] Peter Lawrence Montgomery. *An FFT Extension of the Elliptic Curve Method of Factorization*. PhD thesis, University of California, Los Angeles, 1992.

[MY73] J. Moses and D.Y.Y. Yun. The EZ GCD algorithm. *Proc. ACM Annual Conference*, 73(2):159–166, 1973.

[Mur99] Brian Murphy. *Polynomial selection for the number field sieve integer factorisation algorithm*. PhD thesis, Oxford University, 1999. http://web.comlab.ox.ac.uk/oucl/work/richard.brent/ftp/Murphy-thesis.ps.gz.

[Neu18] Christian Neurohr. *Efficient integration on Riemann surfaces & applications*. Dissertation, Carl von Ossietzky Universität Oldenburg, 2018. URL: http://oops.uni-oldenburg.de/3607.

[PR06] Y.-F. S. Pétermann and Jean-Luc Rémy. Arbitrary Precision Error Analysis for computing $\zeta(s)$ with the Cohen-Olivier algorithm: Complete description of the real case and preliminary report on the general case. Research Report 5852, INRIA, 2006. http://www.inria.fr/rrrt/rr-5852.html.

[PH78] S. C. Pohlig and M. E. Hellman. An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance. *IEEE Trans. Inform. Theory*, 24:106–110, 1978.

[PZ89] Michael E. Pohst and Hans Zassenhaus. *Algorithmic Algebraic Number Theory*. Encyclopaedia of mathematics and its applications. Cambridge University Press, Cambridge, 1989.

[Ros60] Luigi Antonio Rosati. Su una generalizzazione dei piani di Hughes. *Atti Accad. Naz. Lincei Rend. Cl. Sci. Fis. Mat. Nat. (8)*, 29:303–308 (1961), 1960.

[Sch82] A. Schönhage. The fundamental theorem of algebra in terms of computational complexity. Technical Report, Univ. Tübingen, 1982.

[Sch71] Arnold Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.

[Sho95b] Victor Shoup. A New Polynomial Factorization Algorithm and its Implementation. *J. Symbolic Comp.*, 20(4):363–397, 1995.

[SW17] Jonathan Sorenson and Jonathan Webster. Strong Pseudoprimes to Twelve Prime Bases. *Mathematics of Computation*, 86(304):985–1003, 2017. doi:10.1090/mcom/3134.

[Ste05] Allan Steel. Conquering Inseparability: Primary Decomposition and Multivariate Factorization over Algebraic Function Fields of Positive Characteristic. *J. Symbolic Comp.*, 40(3):1053–1075, 2005.

[Tho01] Emmanuel Thomé. Computation of discrete logarithms in $\mathbf{F}_{2^{607}}$. In Colin Boyd and Ed Dawson, editors, *Advances in Cryptology—AsiaCrypt 2001*, volume 2248 of LNCS, 107–124.

Springer-Verlag, 2001. Proc. 7th International Conference on the Theory and Applications of Cryptology and Information Security, Dec. 9–13, 2001, Gold Coast, Queensland, Australia.

[Tra76] Barry M. Trager. Algebraic factoring and rational function integration. In R.D. Jenks, editor, *Proc. SYMSAC '76*, 196–208. ACM press, 1976.

[vH01] Mark van Hoeij. Factoring Polynomials and 0-1 vectors. In *Proceedings of the Cryptography and Lattices Conference (CaLC 2001), Brown University, Providence, RI, USA, March 29-30, 2001*, 142–146. Springer, 2001.

[vH02] Mark van Hoeij. Factoring Polynomials and the knapsack problem. *J. Number Th.*, 95(2):167–189, 2002. http://www.math.fsu.edu/∼hoeij/paper/knapsack.ps.

[vzGG99] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, 1999.

[vzGS92] Joachim von zur Gathen and Victor Shoup. Computing Frobenius Maps And Factoring Polynomials. *Computational Complexity*, 2:187–224, 1992.

[Wan78] Paul S. Wang. An improved multivariate polynomial factoring algorithm. *Math. Comp.*, 32(144):1215–1231, 1978.

[Wan80] Paul S. Wang. The EEZ-GCD algorithm. *SIGSAM Bulletin*, 14(2):50–60, 1980.

[Web95] Kenneth Weber. The Accelerated Integer GCD Algorithm. *ACM Transactions on Mathematical Software*, 21(1):111–122, 1995.

[WW15] E. T. Whittaker and G. N. Watson. *A course of modern analysis*. Cambridge University Press, Cambridge, 2nd edition, 1915.

[Zag] D. Zagier. "Polylogarithms, Dedekind Zeta Functions, and the Algebraic K-Theory of Fields." 377–390.

[Zas35] Hans Zassenhaus. Über endliche Fastkörper. *Abh. Math. Sem. Univ. Hamburg*, 11:187–220, 1935.

Index

Symbols

+ a (*operation*), 7, 68, 83, 97, 110, 133
+ n (*operation*), 16, 54
+ r (*operation*), 156
# F (*operation*), 81
# P (*operation*), 109
# R (*operation*), 5, 53
#N (*operation*), 99
#P (*operation*), 132
- a (*operation*), 7, 68, 83, 97, 111, 133
- n (*operation*), 16, 54
- r (*operation*), 156
- x (*operation*), 38
$0ba_1a_2...a_r$ (*literal*), 13
$0xa_1a_2...a_r$ (*literal*), 13

A

a * b (*operation*), 7, 68, 83, 97, 111, 133
a *:= b (*operation*), 7, 69, 84, 97, 111, 133
a + b (*operation*), 7, 68, 83, 97, 111, 133
a +:= b (*operation*), 7, 69, 83, 97, 111, 133
a . bEcPd (*operation*), 151
a . becpd (*operation*), 151
a / b (*operation*), 7, 66, 69, 83, 97, 111, 133
a /:= b (*operation*), 7, 69
a - b (*operation*), 7, 68, 83, 97, 111, 133
a -:= b (*operation*), 7, 69, 84, 97, 111, 133
a ^ k (*operation*), 7, 69, 83, 97, 111, 133
a ^:= k (*operation*), 8, 69
a div b (*operation*), 111, 133
a div:= b (*operation*), 133
a eq b (*operation*), 8, 69, 84, 98, 111, 133, 155
a ge b (*operation*), 9, 70, 155
a gt b (*operation*), 9, 70, 155
a in F (*operation*), 84
a in I (*operation*), 10, 57, 124
a in N (*operation*), 98
a in R (*operation*), 8, 69, 111, 133, 155
a le b (*operation*), 9, 70, 155
a lt b (*operation*), 9, 70, 155
a mod b (*operation*), 111
a ne b (*operation*), 8, 69, 84, 98, 111, 133, 155
a notin F (*operation*), 84
a notin I (*operation*), 10, 57, 124
a notin N (*operation*), 98
a notin R (*operation*), 8, 69, 111, 134, 155
a.b E cpd (*operation*), 151
a.b e cpd (*operation*), 151
a.bec P d (*operation*), 151
a.bec p d (*operation*), 151
AbelianGroup(G) (*function*), 60
Abs(f) (*function*), 147
Abs(n) (*function*), 19
Abs(p) (*function*), 118
Abs(q) (*function*), 70
Abs(r) (*function*), 159
Abs(x) (*function*), 39
AbsoluteBasis(Q) (*function*), 67
AbsoluteDegree(Q) (*function*), 67
AbsoluteDiscriminant(Q) (*function*), 67
AbsoluteNorm(a) (*function*), 85
AbsoluteTrace(a) (*function*), 85
AbsoluteValue(f) (*function*), 147
AbsoluteValue(n) (*function*), 19
AbsoluteValue(p) (*function*), 118
AbsoluteValue(q) (*function*), 70
AbsoluteValue(r) (*function*), 159
AbsoluteValue(x) (*function*), 39
AdditiveGroup(F) (*function*), 80
AdditiveGroup(R) (*function*), 53
AdditiveGroup(Z) (*function*), 15
AdditiveHilbert90(a, q) (*function*), 86
AffineGroup(GrpPC, N) (*function*), 100
AffineGroup(GrpPerm, N) (*function*), 100
AffineGroup(N) (*function*), 100
AGM(f, g) (*function*), 179
AGM(x, y) (*function*), 179
Algebra(Q, Q) (*function*), 67
AllIrreduciblePolynomials(F, m) (*function*), 87
AllLinearRelations(q,p) (*function*), 164
AllRoots(a, n) (*function*), 86
AllSqrts(a) (*function*), 56
AllSquareRoots(a) (*function*), 56
AlternatingSum(m, i) (*function*), 180
Amicable Pairs of Integers (*example*), 23
Arccos(f) (*function*), 169
Arccos(r) (*function*), 169
Arccosec(r) (*function*), 170
Arccot(r) (*function*), 169
Arcsec(r) (*function*), 169
Arcsin(f) (*function*), 169
Arcsin(r) (*function*), 169
Arctan(f) (*function*), 169
Arctan(r) (*function*), 169
Arctan(x, y) (*function*), 169
Arctan2(x, y) (*function*), 169
Arg(c) (*function*), 157

Argcosech(r) (*function*), 171
Argcosech(s) (*function*), 171
Argcosh(f) (*function*), 171
Argcosh(r) (*function*), 171
Argcoth(r) (*function*), 171
Argcoth(s) (*function*), 171
Argsech(r) (*function*), 171
Argsech(s) (*function*), 171
Argsinh(f) (*function*), 171
Argsinh(r) (*function*), 171
Argtanh(f) (*function*), 171
Argtanh(r) (*function*), 171
Argtanh(s) (*function*), 171
Argument of a Signed Zero (*example*), 157
Argument(c) (*function*), 157
ArithmeticGeometricMean(x, y) (*function*), 179
AssertAttribute(F, "PowerPrinting", l) (*procedure*), 77
AssertAttribute(FldFin, "PowerPrinting", l) (*procedure*), 77
AssertAttribute(RngInt, "CunninghamStorageLimit", l) (*procedure*), 31
Assigning Generator Names (*example*), 129
AssignNames(~C, [s]) (*procedure*), 150
AssignNames(~F, [f]) (*procedure*), 77
AssignNames(~G, S) (*procedure*), 59
AssignNames(~P, s) (*procedure*), 106, 130
AssociatedPrimitiveCharacter(chi) (*function*), 61
Automatic Coercion of Rationals (*example*), 64
AutomorphismGroup(K, k) (*function*), 81
AutomorphismGroup(N) (*function*), 101
AutomorphismGroup(Q) (*function*), 67
AutomorphismGroup(Q, Q) (*function*), 67
$a_1a_2...a_r$ (*literal*), 13

B

BaseExtend(G, R) (*function*), 59
BaseExtend(G, R, z) (*function*), 59
BaseField(F) (*function*), 75
BaseField(Q) (*function*), 67
BaseMPolynomial(n, m, d) (*function*), 46
BaseRing(chi) (*function*), 60
BaseRing(G) (*function*), 60
BaseRing(P) (*function*), 108, 131
Basis(Q) (*function*), 67
Bell(n) (*function*), 24
BernoulliApproximation(n) (*function*), 179
BernoulliNumber(n) (*function*), 179
BernoulliPolynomial(n) (*function*), 126
BesselFunction(n, r) (*function*), 177
BesselFunctionSecondKind(n, r) (*function*), 178
BestApproximation(r, n) (*function*), 164
BestTranslation(F, m, a) (*function*), 48
Binomial(n, r) (*function*), 24
BitPrecision(c) (*function*), 158
BitPrecision(R) (*function*), 154
BitPrecision(r) (*function*), 158
BitwiseAnd(x, y) (*function*), 17
BitwiseNot(x) (*function*), 17
BitwiseOr(x, y) (*function*), 17
BitwiseXor(x, y) (*function*), 17

C

C ! a (*operation*), 152
C ! [x, y] (*operation*), 151
Cardinality(N) (*function*), 99
CarmichaelLambda(n) (*function*), 22
CarmichaelLambda(Q) (*function*), 22
Catalan(R) (*function*), 158
Category(a) (*function*), 84, 98
Category(F) (*function*), 80
Category(n) (*function*), 18, 55
Category(P) (*function*), 108, 131
Category(p) (*function*), 110
Category(Q) (*function*), 66
Category(R) (*function*), 5, 53, 153
Category(r) (*function*), 6, 68, 154
Category(Z) (*function*), 15
Ceiling(n) (*function*), 20
Ceiling(q) (*function*), 71
Ceiling(r) (*function*), 158
Ceiling(x) (*function*), 39
Center(R) (*function*), 5, 53
Center(Z) (*function*), 15
Centre(F) (*function*), 80
Centre(R) (*function*), 5
ChangePrecision(c, n) (*function*), 158
ChangePrecision(r, n) (*function*), 158
ChangeRing(P, S) (*function*), 108, 132
ChangeRing(P, S, f) (*function*), 108
Changing the Coefficient Ring (*example*), 108
Characteristic(F) (*function*), 81
Characteristic(P) (*function*), 109, 132
Characteristic(Q) (*function*), 67
Characteristic(R) (*function*), 5, 53, 153
Characteristic(Z) (*function*), 15
CharacteristicPolynomial(a) (*function*), 85
CharacteristicPolynomial(a, E) (*function*), 85
ChebyshevFirst(n) (*function*), 125
ChebyshevSecond(n) (*function*), 125
ChebyshevT(n) (*function*), 125
ChebyshevU(n) (*function*), 125
CheckCertificate(cert) (*function*), 25
chi(n) (*function*), 61
ChineseRemainderTheorem(I, J, a, b) (*function*), 51
ChineseRemainderTheorem(X, M) (*function*), 116
ChineseRemainderTheorem(X, N) (*function*), 36
ClassGroup(Q) (*function*), 67
ClassGroup(Z) (*function*), 15
ClassRepresentative(I) (*function*), 51
ClearDenominators(f) (*function*), 142

ClearDenominators(Q) (*function*), 142
ClearStoredFactors() (*procedure*), 30
ClenshawCurtisIntegrationPoints(N,D) (*function*), 182
Coefficient(f, i, k) (*function*), 134
Coefficient(f, v, k) (*function*), 134
Coefficient(p, i) (*function*), 112
CoefficientDenominator(f) (*function*), 135
CoefficientNumerator(f) (*function*), 135
CoefficientRing(f) (*function*), 108
CoefficientRing(P) (*function*), 108, 131
Coefficients and Monomials (*example*), 137
Coefficients(f) (*function*), 134
Coefficients(f, i) (*function*), 134
Coefficients(f, v) (*function*), 134
Coefficients(p) (*function*), 112
CoefficientsAndMonomials(f) (*function*), 135
Coercion between Residue Class Rings (*example*), 52
CommonOverfield(K, L) (*function*), 76
comp< R | a_1, ..., a_r > (*constructor*), 11
CompanionMatrix(f) (*function*), 122
ComplementaryErrorFunction(r) (*function*), 179
Completion(Q, P) (*function*), 65
Completion(R, P) (*function*), 11
ComplexConjugate(c) (*function*), 159
ComplexConjugate(n) (*function*), 19
ComplexConjugate(q) (*function*), 70
ComplexConjugate(r) (*function*), 159
ComplexField() (*function*), 151
ComplexField(p) (*function*), 150
ComplexField(R) (*function*), 151
ComplexToPolar(c) (*function*), 157
Conductor(chi) (*function*), 60
Conductor(Q) (*function*), 67
Conjugate(c) (*function*), 159
Conjugate(n) (*function*), 19
Conjugate(q) (*function*), 70
ConstantCoefficient(p) (*function*), 112
Constructing Dickson Nearfields (*example*), 96
Content(f) (*function*), 143
Content(p) (*function*), 118
ContentAndPrimitivePart(f) (*function*), 143
ContentAndPrimitivePart(p) (*function*), 118
ContinuedFraction(r) (*function*), 71, 163
ContinuedFractionValue(C) (*function*), 71
Contpp(f) (*function*), 143
Contpp(p) (*function*), 118
Convergents(s) (*function*), 164
Converting to a Univariate Polynomial (*example*), 139
ConvertToCWIFormat(P, pb) (*function*), 46
ConwayPolynomial(p, n) (*function*), 87
CoprimeBasis(S) (*function*), 33
Cos(c) (*function*), 168
Cos(f) (*function*), 167
Cos(r) (*function*), 168
Cosec(c) (*function*), 168
Cosec(f) (*function*), 168
Cosec(r) (*function*), 168
Cosech(r) (*function*), 170
Cosh(f) (*function*), 170
Cosh(r) (*function*), 170
Cot(c) (*function*), 168
Cot(f) (*function*), 168
Cot(r) (*function*), 168
Coth(r) (*function*), 170
CreateCharacterFile(P) (*function*), 43
CreateCharacterFile(P, cc) (*function*), 43
CreateCycleFile(P) (*function*), 43
Creating a Complex Field (*example*), 151
Creating Integers (*example*), 13
Creating Polynomials (*example*), 107
Creating Real Numbers (*example*), 152
CRT(X, M) (*function*), 116
CRT(X, N) (*function*), 36
Cube Root in a Prime Residue Ring (*example*), 52
Cunningham(b, k, c) (*function*), 30
CycleCount(fn) (*function*), 43
CycleCount(P) (*function*), 43
Cyclic Unit Group (*example*), 58

D

DawsonIntegral(r) (*function*), 179
Decomposition(chi) (*function*), 61
Decomposition(f) (*function*), 115
Decomposition(Q, p) (*function*), 67
Decomposition(R, p) (*function*), 51
DedekindEta(s) (*function*), 174
DedekindEta(z) (*function*), 174
DedekindTest(p, m) (*function*), 118
DefiningPolynomial(F) (*function*), 81
DefiningPolynomial(F, E) (*function*), 81
DefiningPolynomial(Q) (*function*), 67
Degree(F) (*function*), 81
Degree(F, E) (*function*), 81
Degree(f, i) (*function*), 138
Degree(f, v) (*function*), 138
Degree(I) (*function*), 51
Degree(p) (*function*), 112
Degree(Q) (*function*), 67
Delta(L) (*function*), 175
Delta(t) (*function*), 175
Delta(z) (*function*), 174
Denominator(n) (*function*), 15
Denominator(q) (*function*), 69
Derivative(f, i) (*function*), 139
Derivative(f, k, i) (*function*), 139
Derivative(f, k, v) (*function*), 139
Derivative(f, v) (*function*), 139
Derivative(p) (*function*), 114
Derivative(p, n) (*function*), 114
DiagonalForm(f) (*function*), 141
Diameter(L) (*function*), 160

DickmanRho(u) (*function*), 22
Dickson Pairs (*example*), 95
Dickson Permutation Polynomials (*example*), 91
DicksonFirst(n, a) (*function*), 90, 126
DicksonNearfield(q, v : parameters) (*function*), 96
DicksonPairs(p, h1, v1) (*function*), 95
DicksonPairs(p, hlo, hhi, vlo, vhi) (*function*), 95
DicksonSecond(n, a) (*function*), 91, 126
DicksonTriples(p, hb, vb) (*function*), 95
Dilog(s) (*function*), 166
Dirichlet Characters Modulo 5 (*example*), 62
DirichletGroup(N) (*function*), 59
DirichletGroup(N,R) (*function*), 59
DirichletGroup(N,R,z,r) (*function*), 59
Discrete Logarithms in a Finite Field (*example*), 89
DiscreteFourierTransform(E) (*function*), 181
Discriminant(f) (*function*), 122
Discriminant(f, i) (*function*), 146
Discriminant(f, v) (*function*), 146
Discriminant(Q) (*function*), 67
Distance(x, L) (*function*), 159
DistinctDegreeFactorization(f) (*function*), 122
Distributed NFS Factorization (*example*), 45
Divisors(f) (*function*), 33, 35
Divisors(n) (*function*), 33
DivisorSigma(i, n) (*function*), 22
DivisorSigma(i, Q) (*function*), 22

E

ECM(n, B1) (*function*), 32
ECMFactoredOrder(p, s) (*function*), 33
ECMOrder(p, s) (*function*), 33
ECMSteps(n, L, U) (*function*), 33
Eisenstein(k, F) (*function*), 172
Eisenstein(k, L) (*function*), 172
Eisenstein(k, t) (*function*), 172
Eisenstein(k, z) (*function*), 172
Element(N, x) (*function*), 98
Elements(G) (*function*), 59
ElementToSequence(a) (*function*), 72, 79
ElementToSequence(a, E) (*function*), 79
ElementToSequence(chi) (*function*), 60
ElementToSequence(f) (*function*), 35
ElementToSequence(p) (*function*), 112
ElementToSequence(x) (*function*), 98
elt< P | a_0, ..., a_d > (*constructor*), 107
elt< P | s > (*constructor*), 107
elt< Q | a, b > (*constructor*), 66
elt< R | a > (*constructor*), 131
elt< R | k > (*constructor*), 54
elt< R | s > (*constructor*), 131
elt< Z | 0b$a_1a_2...a_r$ > (*constructor*), 13
elt< Z | 0x$a_1a_2...a_r$ > (*constructor*), 13
elt< Z | $a_1a_2...a_r$ > (*constructor*), 13
elt<C | x, y> (*constructor*), 151
elt<F | a> (*constructor*), 78
elt<F | a_0, ..., a_{n-1}> (*constructor*), 78
elt<R | m, n> (*constructor*), 151
Eltseq(a) (*function*), 72, 79
Eltseq(a, E) (*function*), 79
Eltseq(f) (*function*), 35
Eltseq(n) (*function*), 15
Eltseq(p) (*function*), 112
Embed(E, F) (*procedure*), 76
Embed(E, F, x) (*procedure*), 77
Embedding a Real Field in a Polynomial Ring (*example*), 149
EqualDegreeFactorization(f, d, g) (*function*), 122
Erf(r) (*function*), 179
Erfc(r) (*function*), 179
ErrorFunction(r) (*function*), 179
EuclideanNorm(n) (*function*), 19
EuclideanNorm(p) (*function*), 116
EulerGamma(R) (*function*), 158
EulerPhi(n) (*function*), 22
EulerPhi(Q) (*function*), 22
EulerPhiInverse(m) (*function*), 22
EulerPhiInverse(Q) (*function*), 22
Evaluate(chi,n) (*function*), 61
Evaluate(f, i, r) (*function*), 140
Evaluate(f, s) (*function*), 140
Evaluate(f, v, r) (*function*), 140
Evaluate(p, r) (*function*), 114
ExactQuotient(f, g) (*function*), 116, 140
ExactQuotient(n, d) (*function*), 17
ExistsConwayPolynomial(p, n) (*function*), 87
Exp(c) (*function*), 166
Exp(f) (*function*), 166
Exp(r) (*function*), 166
Exponent(G) (*function*), 60
ExponentialIntegral(r) (*function*), 179
ExponentialIntegralE1(r) (*function*), 179
Exponents(f) (*function*), 136
ext< R | > (*constructor*), 11
ext< R, n | > (*constructor*), 11
ext<F | n> (*constructor*), 75
ext<F | P> (*constructor*), 75
ExtendedGreatestCommonDivisor(f, g) (*function*), 117
ExtendedGreatestCommonDivisor(m, n) (*function*), 21
ExtendedGreatestCommonDivisor(s) (*function*), 22
ExtendedUnitGroup(D) (*function*), 101
ExtensionField<F, x | P> (*constructor*), 75

F

F ! a (*operation*), 78
F . 1 (*operation*), 78, 79
f ^ M (*operation*), 116

f div g (*operation*), 115, 140
F eq G (*operation*), 82
F meet G (*operation*), 76
f mod g (*operation*), 116
F ne G (*operation*), 82
Facint(f) (*function*), 34
Facint(s) (*function*), 14
Facpol(f) (*function*), 120
Factor(P) (*function*), 44
Factor(P,k) (*function*), 44
FactoredCarmichaelLambda(n) (*function*), 22
FactoredCarmichaelLambda(Q) (*function*), 22
FactoredEulerPhi(n) (*function*), 22
FactoredEulerPhi(Q) (*function*), 22
FactoredEulerPhiInverse(n) (*function*), 23
FactoredEulerPhiInverse(Q) (*function*), 23
FactoredModulus(R) (*function*), 53
FactoredOrder(a) (*function*), 86
Factorial(n) (*function*), 24
Factoring an RSA Modulus with Small Roots (*example*), 113
Factoring Swinnerton-Dyer Polynomials (*example*), 120
Factorisation(f) (*function*), 119
Factorisation(n) (*function*), 29
Factorisation(n: parameters) (*function*), 29
FactorisationOverSplittingField(f) (*function*), 82
FactorisationToInteger(s) (*function*), 14
FactorisationToPolynomial(f) (*function*), 120
Factorization(f) (*function*), 119, 143
Factorization(n) (*function*), 29
Factorization(n: parameters) (*function*), 29
FactorizationOverSplittingField(f) (*function*), 82
FactorizationToInteger(f) (*function*), 34
FactorizationToInteger(s) (*function*), 14
Factorizing a Product of Trinomials (*example*), 144
Factorizing a Vandermonde Determinant (*example*), 144
Failure of Left Distributivity (*example*), 99
Fibonacci(n) (*function*), 24
FieldOfFractions(F) (*function*), 80
FieldOfFractions(Q) (*function*), 65
FieldOfFractions(Z) (*function*), 15, 65
FindDependencies(P) (*function*), 43
Finding a Good NFS Polynomial (*example*), 47
FindRelations(P) (*function*), 43
FindRelationsInCWIFormat(P) (*function*), 46
Finite Field as a Vector Space (*example*), 81
Finite Field Factorization (*example*), 145
FiniteField(p, n) (*function*), 74
FiniteField(q) (*function*), 74
Floor(n) (*function*), 20
Floor(q) (*function*), 71
Floor(r) (*function*), 158
Floor(x) (*function*), 39
Frobenius(a) (*function*), 85
Frobenius(a, E) (*function*), 85
Frobenius(a, E, r) (*function*), 85
Frobenius(a, r) (*function*), 85
FullDirichletGroup(N) (*function*), 59

G

G ! x (*operation*), 59
G . i (*operation*), 59
GaloisConjugacyRepresentatives(G) (*function*), 60
GaloisConjugacyRepresentatives(seq) (*function*), 60
GaloisField(p, n) (*function*), 74
GaloisField(q) (*function*), 74
GaloisGroup(K, k) (*function*), 81
Gamma(f) (*function*), 176
Gamma(r) (*function*), 176
Gamma(s, t) (*function*), 177
GammaD(s) (*function*), 177
GaussJacobiIntegrationPoints(N,D,a,b) (*function*), 181
GaussLegendreIntegrationPoints(N,D) (*function*), 181
GCD(a, b) (*function*), 56
Gcd(a, b) (*function*), 56
GCD(f, g) (*function*), 117, 142
Gcd(f, g) (*function*), 117, 142
GCD(m, n) (*function*), 21
Gcd(m, n) (*function*), 21
GCD(Q) (*function*), 57, 142
Gcd(Q) (*function*), 57
GCD(s) (*function*), 21
Gcd(s) (*function*), 21
Gcd(s, t) (*function*), 35
GegenbauerPolynomial(n, m) (*function*), 125
GeneralizedFibonacciNumber(g0, g1, n) (*function*), 24
Generator(F) (*function*), 79
Generator(F, E) (*function*), 79
Generator(I) (*function*), 51
Generators(G) (*function*), 60
GetDefaultRealField() (*function*), 150
GetGMPVersion() (*procedure*), 150
GetMPCVersion() (*procedure*), 150
GetMPFRVersion() (*procedure*), 150
GetStoredFactors() (*function*), 30
GF(p, n) (*function*), 74
GF(q) (*function*), 74
Global and Non-Global Polynomial Rings (*example*), 105, 129
GreatestCommonDivisor(a, b) (*function*), 56
GreatestCommonDivisor(f, g) (*function*), 117, 142
GreatestCommonDivisor(m, n) (*function*), 21
GreatestCommonDivisor(Q) (*function*), 57
GreatestCommonDivisor(s) (*function*), 21
GroundField(F) (*function*), 75

H

HasAttribute(F, "PowerPrinting") (*function*), 77
HasAttribute(FldFin, "PowerPrinting", l) (*procedure*), 77
HasGCD(R) (*function*), 6
HasPolynomialFactorization(R) (*function*), 120
HasRoot(p) (*function*), 113
HasRoot(p, S) (*function*), 113
Height(q) (*function*), 70
Hensel Lifting (*example*), 123
HenselLift(f, R, k) (*function*), 163
HenselLift(f, s, P) (*function*), 123
HermitePolynomial(n) (*function*), 125
Heron's Formula for the Area of a Triangle (*example*), 145
Hilbert90(a, q) (*function*), 86
HirzebruchJungContinuedFraction(r) (*function*), 71
HirzebruchJungContinuedFractionValue(C) (*function*), 71
HJContinuedFraction(r) (*function*), 71
HJContinuedFractionValue(C) (*function*), 71
hom< F -> G | x > (*constructor*), 78
hom< P -> S | f, y > (*constructor*), 110
hom< P -> S | f, y_1, ..., y_n > (*constructor*), 132
hom< P -> S | y > (*constructor*), 110
hom< P -> S | y_1, ..., y_n > (*constructor*), 132
hom< Q -> R | > (*constructor*), 65
hom< R -> S | > (*constructor*), 54, 149
hom< Z -> R | > (*constructor*), 12
Homomorphism from the Integers to a Matrix Ring (*example*), 12
Homomorphism into a Number Field (*example*), 133
Homomorphism into the Real Field (*example*), 110
Hughes Planes (*example*), 102
HughesPlane(N : parameters) (*function*), 102
HypergeometricSeries(a, b, c, z) (*function*), 178
HypergeometricU(a, b, s) (*function*), 178

I

I * J (*operation*), 10, 57, 124
I + J (*operation*), 10, 57, 124
I . 1 (*operation*), 124
I eq J (*operation*), 10, 57, 124
I meet J (*operation*), 10, 57, 124
I ne J (*operation*), 10, 57, 124
I notsubset J (*operation*), 10, 57, 124
I subset J (*operation*), 10, 57, 124
Id(R) (*function*), 7
ideal< R | a > (*constructor*), 50
ideal< R | a_1, ..., a_r > (*constructor*), 9, 56, 123
Ideals of the Integers (*example*), 50
Identity(F) (*function*), 78
Identity(N) (*function*), 99
Identity(P) (*function*), 107, 131
Identity(Q) (*function*), 66
Identity(R) (*function*), 54, 152
Identity(Z) (*function*), 14
Ilog(b, n) (*function*), 19
Ilog2(n) (*function*), 19
Im(c) (*function*), 157
Imaginary(c) (*function*), 157
InfiniteSum(m, i) (*function*), 180
Infinity() (*function*), 37
IntegerRelation(q) (*function*), 164
IntegerRelation(q,N) (*function*), 164
IntegerRing() (*function*), 13, 65
IntegerRing(m) (*function*), 52
IntegerRing(Q) (*function*), 13, 52, 65
Integers() (*function*), 13, 65
Integers(m) (*function*), 52
Integers(Q) (*function*), 13, 52
IntegerToSequence(n, b) (*function*), 14
IntegerToString(n) (*function*), 15
IntegerToString(n, b) (*function*), 15
Integral over an Infinite Interval (*example*), 183
Integral with Endpoint Singularities (*example*), 183
Integral(f, i) (*function*), 139
Integral(f, v) (*function*), 139
Integral(p) (*function*), 114
IntegralBasis(Q) (*function*), 66
Interpolation(I, V) (*function*), 114
Interpolation(I, V, i) (*function*), 140
Interpolation(I, V, v) (*function*), 140
Interpolation(P, V, t) (*function*), 181
Intseq(n, b) (*function*), 14
Inverse(a) (*function*), 97
InverseMod(n, m) (*function*), 36
Iroot(a, n) (*function*), 20
IrreducibleLowTermGF2Polynomial(n) (*function*), 87
IrreduciblePolynomial(F, n) (*function*), 87
IrreducibleSparseGF2Polynomial(n) (*function*), 87
IsAlgebraicallyDependent(S) (*function*), 134
IsCommutative(F) (*function*), 82
IsCommutative(P) (*function*), 109, 132
IsCommutative(Q) (*function*), 68
IsCommutative(R) (*function*), 5, 53, 153
IsCommutative(Z) (*function*), 15
IsConway(F) (*function*), 82
IsDefault(F) (*function*), 82
IsDivisibleBy(a, b) (*function*), 134
IsDivisibleBy(f, g) (*function*), 116
IsDivisibleBy(n, d) (*function*), 18
IsDivisionRing(F) (*function*), 82
IsDivisionRing(P) (*function*), 109, 132
IsDivisionRing(Q) (*function*), 68

IsDivisionRing(R) (*function*), 5, 54, 153
IsDivisionRing(Z) (*function*), 16
IsDomain(F) (*function*), 82
IsDomain(P) (*function*), 109, 132
IsDomain(Q) (*function*), 68
IsDomain(R) (*function*), 6, 54, 154
IsDomain(Z) (*function*), 16
IsEuclideanDomain(F) (*function*), 82
IsEuclideanDomain(P) (*function*), 109, 132
IsEuclideanDomain(Q) (*function*), 68
IsEuclideanDomain(R) (*function*), 5, 54, 153
IsEuclideanDomain(Z) (*function*), 16
IsEuclideanRing(F) (*function*), 82
IsEuclideanRing(P) (*function*), 109, 132
IsEuclideanRing(Q) (*function*), 68
IsEuclideanRing(R) (*function*), 6, 54, 154
IsEuclideanRing(Z) (*function*), 16
IsEven(chi) (*function*), 61
IsEven(n) (*function*), 18
IsEven(s) (*function*), 35
IsFactorial(n) (*function*), 24
IsField(F) (*function*), 82
IsField(P) (*function*), 109, 132
IsField(Q) (*function*), 68
IsField(R) (*function*), 5, 53, 153
IsField(Z) (*function*), 16
IsFinite(F) (*function*), 82
IsFinite(P) (*function*), 109, 132
IsFinite(Q) (*function*), 68
IsFinite(R) (*function*), 5, 53, 153
IsFinite(x) (*function*), 39
IsFinite(Z) (*function*), 15
IsIdempotent(a) (*function*), 69, 84, 111
IsIdempotent(f) (*function*), 134
IsIdempotent(n) (*function*), 19, 55
IsIdempotent(r) (*function*), 155
IsIdempotent(x) (*function*), 8
IsIdentity(a) (*function*), 98
IsIntegral(c) (*function*), 156
IsIntegral(n) (*function*), 18
IsIntegral(q) (*function*), 69
IsIntegralDomain(R) (*function*), 6
IsIrreducible(a) (*function*), 70, 84, 111
IsIrreducible(f) (*function*), 122, 134, 144
IsIrreducible(n) (*function*), 19, 55
IsIrreducible(r) (*function*), 155
IsIrreducible(x) (*function*), 8
IsIsomorphic(E, F) (*function*), 77
IsIsomorphic(N1, N2) (*function*), 101
IsMagmaEuclideanRing(R) (*function*), 6
IsMinusOne(a) (*function*), 8, 69, 84, 111
IsMinusOne(f) (*function*), 134
IsMinusOne(n) (*function*), 18, 55
IsMinusOne(r) (*function*), 154
IsMonic(a) (*function*), 111
IsNilpotent(a) (*function*), 69, 84, 111
IsNilpotent(f) (*function*), 134
IsNilpotent(n) (*function*), 18, 55
IsNilpotent(r) (*function*), 155
IsNilpotent(x) (*function*), 8
IsNormal(a) (*function*), 84
IsNormal(a, E) (*function*), 84
IsOdd(chi) (*function*), 61
IsOdd(n) (*function*), 18
IsOdd(s) (*function*), 35
IsOne(a) (*function*), 8, 69, 84, 111
IsOne(f) (*function*), 134
IsOne(n) (*function*), 18, 55
IsOne(r) (*function*), 154
IsOne(s) (*function*), 35
IsOrdered(F) (*function*), 82
IsOrdered(P) (*function*), 109, 132
IsOrdered(Q) (*function*), 68
IsOrdered(R) (*function*), 5, 53, 153
IsOrdered(Z) (*function*), 16
IsPID(F) (*function*), 82
IsPID(P) (*function*), 109, 132
IsPID(Q) (*function*), 68
IsPID(R) (*function*), 6, 54, 153
IsPID(Z) (*function*), 16
IsPIR(R) (*function*), 6
IsPower(a, n) (*function*), 86
IsPower(n) (*function*), 18
IsPower(n, k) (*function*), 18
IsPrime(a) (*function*), 70, 84, 111
IsPrime(f) (*function*), 134
IsPrime(n) (*function*), 18, 19, 25, 55
IsPrime(n: parameter) (*function*), 25
IsPrime(r) (*function*), 155
IsPrime(s) (*function*), 35
IsPrime(x) (*function*), 8
IsPrimeField(F) (*function*), 76
IsPrimePower(n) (*function*), 26
IsPrimePower(s) (*function*), 35
IsPrimitive(a) (*function*), 84
IsPrimitive(chi) (*function*), 60
IsPrimitive(f) (*function*), 84
IsPrimitive(n) (*function*), 57
IsPrimitive(n, m) (*function*), 36
IsPrincipalIdealDomain(R) (*function*), 6
IsPrincipalIdealRing(F) (*function*), 82
IsPrincipalIdealRing(P) (*function*), 109, 132
IsPrincipalIdealRing(Q) (*function*), 68
IsPrincipalIdealRing(R) (*function*), 6, 54, 154
IsPrincipalIdealRing(Z) (*function*), 16
IsProbablePrime(n: parameter) (*function*), 26
IsProbablyPermutationPolynomial(p) (*function*), 91
IsProbablyPrime(n: parameter) (*function*), 26
Isqrt(n) (*function*), 20
IsReal(c) (*function*), 156
IsRegular(a) (*function*), 70, 84, 111
IsRegular(f) (*function*), 134
IsRegular(n) (*function*), 19, 55
IsSeparable(f) (*function*), 122
IsSinglePrecision(n) (*function*), 18

IsSquare(a) (*function*), 84
IsSquare(n) (*function*), 18, 56
IsSquare(s) (*function*), 36
IsSquarefree(n) (*function*), 18
IsSquarefree(s) (*function*), 36
IsTotallyEven(chi) (*function*), 61
IsTrivial(chi) (*function*), 60
IsUFD(F) (*function*), 82
IsUFD(P) (*function*), 109, 132
IsUFD(Q) (*function*), 68
IsUFD(R) (*function*), 6, 54, 153
IsUFD(Z) (*function*), 16
IsUniqueFactorizationDomain(R) (*function*), 6
IsUnit(a) (*function*), 8, 69, 84, 98, 111
IsUnit(f) (*function*), 134
IsUnit(n) (*function*), 19, 55
IsUnit(r) (*function*), 155
IsUnit(s) (*function*), 35
IsUnitary(F) (*function*), 82
IsUnitary(P) (*function*), 109, 132
IsUnitary(Q) (*function*), 68
IsUnitary(R) (*function*), 5, 53, 153
IsUnitary(Z) (*function*), 15
IsUnivariate(f) (*function*), 138
IsUnivariate(f, i) (*function*), 138
IsUnivariate(f, v) (*function*), 138
IsZero(a) (*function*), 8, 69, 84, 98, 111
IsZero(f) (*function*), 134
IsZero(n) (*function*), 18, 55
IsZero(r) (*function*), 154
IsZeroDivisor(a) (*function*), 69, 84, 111
IsZeroDivisor(f) (*function*), 134
IsZeroDivisor(n) (*function*), 19, 55
IsZeroDivisor(r) (*function*), 155
IsZeroDivisor(x) (*function*), 8

J

JacobianMatrix([f]) (*function*), 140
JacobiSymbol(a,b) (*function*), 119
JacobiSymbol(n, m) (*function*), 23
JacobiTheta(q, z) (*function*), 173, 174
JacobiThetaNullK(q, k) (*function*), 174
JBessel(n, s) (*function*), 178
jInvariant(F) (*function*), 174
jInvariant(L) (*function*), 174
jInvariant(q) (*function*), 174
jInvariant(s) (*function*), 174

K

KBessel(n, s) (*function*), 178
KBessel2(n, s) (*function*), 178
Kernel(N) (*function*), 99
KroneckerCharacter(D) (*function*), 59
KroneckerCharacter(D, R) (*function*), 59
KroneckerSymbol(n, m) (*function*), 23

L

LaguerrePolynomial(n) (*function*), 125
LaguerrePolynomial(n, m) (*function*), 125
LCM(a, b) (*function*), 57
Lcm(a, b) (*function*), 57
LCM(f, g) (*function*), 117, 142
Lcm(f, g) (*function*), 117, 142
LCM(m, n) (*function*), 22
Lcm(m, n) (*function*), 22
LCM(Q) (*function*), 57, 142
Lcm(Q) (*function*), 57
LCM(s) (*function*), 22
Lcm(s) (*function*), 22
Lcm(s, t) (*function*), 35
LeadingCoefficient(f) (*function*), 135
LeadingCoefficient(f, i) (*function*), 135
LeadingCoefficient(f, v) (*function*), 135
LeadingCoefficient(p) (*function*), 112
LeadingMonomial(f) (*function*), 136
LeadingTerm(f) (*function*), 136
LeadingTerm(f, i) (*function*), 136
LeadingTerm(f, v) (*function*), 136
LeadingTerm(p) (*function*), 112
LeadingTotalDegree(f) (*function*), 138
LeastCommonMultiple(a, b) (*function*), 57
LeastCommonMultiple(f, g) (*function*), 117, 142
LeastCommonMultiple(m, n) (*function*), 22
LeastCommonMultiple(Q) (*function*), 57
LeastCommonMultiple(s) (*function*), 22
LegendrePolynomial(n) (*function*), 125
LegendreSymbol(n, m) (*function*), 23
Length(f) (*function*), 135
LinearRelation(q: parameters) (*function*), 164
LinearRelation(v: parameters) (*function*), 164
loc< R | a_1, ..., a_r > (*constructor*), 11
Localization(R, P) (*function*), 11
Log(b, r) (*function*), 166
Log(b, x) (*function*), 89
Log(c) (*function*), 166
Log(f) (*function*), 166
Log(r) (*function*), 166
Log(x) (*function*), 89
LogDerivative(s) (*function*), 177
LogGamma(f) (*function*), 177
LogGamma(r) (*function*), 177
LogIntegral(r) (*function*), 180
Lucas(n) (*function*), 24

M

m * n (*operation*), 16, 55
m *:= n (*operation*), 16, 55
m + n (*operation*), 16, 54
m +:= n (*operation*), 16, 55
m / n (*operation*), 16, 55
m /:= n (*operation*), 16, 55
m - n (*operation*), 16, 54
m -:= n (*operation*), 16, 55
m ^:= k (*operation*), 16, 55
m div n (*operation*), 55
m eq n (*operation*), 17, 55

m ge n (*operation*), 19
m gt n (*operation*), 19
m le n (*operation*), 19
m lt n (*operation*), 19
m ne n (*operation*), 17, 55
MantissaExponent(r) (*function*), 157
Mapping Rationals into the Integers Modulo 11 (*example*), 65
MatrixAlgebra(A, E) (*function*), 81
MatrixAlgebra(F, E) (*function*), 80
MaximalOrder(Q) (*function*), 65
Maximum(a, b) (*function*), 9, 70, 155
Maximum(m, n) (*function*), 19
Maximum(Q) (*function*), 9, 19, 70, 155
Maximum(x, y) (*function*), 39
MaxNorm(f) (*function*), 147
MaxNorm(p) (*function*), 118
MergeFiles(S, fn) (*function*), 44
MinimalBaseRingCharacter(chi) (*function*), 61
MinimalField(q) (*function*), 66
MinimalField(S) (*function*), 67
MinimalPolynomial(a) (*function*), 85
MinimalPolynomial(a, E) (*function*), 85
MinimalPolynomial(n) (*function*), 19
MinimalPolynomial(q) (*function*), 70
MinimalPolynomial(r,d) (*function*), 165
MinimalPolynomial(r,d,N) (*function*), 165
Minimum(a, b) (*function*), 9, 70, 155
Minimum(m, n) (*function*), 19
Minimum(Q) (*function*), 9, 19, 70, 155
Minimum(x, y) (*function*), 39
MinusInfinity() (*function*), 37
ModByPowerOf2(n, b) (*function*), 17
Modexp(f, n, g) (*function*), 116
Modexp(n, k, m) (*function*), 36
Modinv(n, m) (*function*), 36
Modorder(n, m) (*function*), 36
Modsqrt(n, m) (*function*), 36
Modulus(c) (*function*), 157
Modulus(chi) (*function*), 60
Modulus(G) (*function*), 60
Modulus(Q) (*function*), 125
Modulus(R) (*function*), 53
MoebiusMu(f) (*function*), 35
MoebiusMu(n) (*function*), 23
MoebiusMu(Q) (*function*), 23
Monomial(P, E) (*function*), 137
MonomialCoefficient(f, m) (*function*), 135
MonomialCoefficient(p, m) (*function*), 112
Monomials(f) (*function*), 135
Monomials(p) (*function*), 112
MPQS(n) (*function*), 33
Multinomial(n, [r_1, ... r_n]) (*function*), 24
MultiplicativeGroup(F) (*function*), 80
MultiplicativeGroup(R) (*function*), 53
MultiplicativeGroup(Z) (*function*), 15
MultivariatePolynomial(P, f, i) (*function*), 131
MultivariatePolynomial(P, f, v) (*function*), 131
MurphyAlphaApproximation(F, b) (*function*), 46

N

N ! x (*operation*), 98
n ^ k (*operation*), 16, 55
n div m (*operation*), 16
n div:= m (*operation*), 17
N eq M (*operation*), 99
n in R (*operation*), 17, 55
n mod m (*operation*), 16, 36
n mod:= m (*operation*), 17
N ne M (*operation*), 99
n notin R (*operation*), 17, 55
Name(C, 1) (*function*), 150
Name(F, 1) (*function*), 78
Name(P, i) (*function*), 106, 130
NextPrime(n) (*function*), 27
NextPrime(n: parameter) (*function*), 27
NFS Attributes for a 70-Digit Number (*example*), 41
NFS Attributes for an 80-Digit Number (*example*), 41
NFS Attributes for an 87-Digit Number (*example*), 42
NFS Process Attributes (*example*), 41
NFS(n, F, m1, m2) (*function*), 40
NFSProcess(n, F, m1, m2) (*function*), 41
Norm Equation (*example*), 37
Norm(a) (*function*), 85
Norm(a, E) (*function*), 85
Norm(c) (*function*), 159
Norm(n) (*function*), 19
Norm(q) (*function*), 70
Norm(r) (*function*), 159
NormAbs(a) (*function*), 85
NormalElement(F) (*function*), 79
NormalElement(F, E) (*function*), 79
Normalise(x) (*function*), 57
Normalize(f) (*function*), 117, 142
Normalize(x) (*function*), 57
NormEquation(d, m) (*function*), 37
NormEquation(d, m: parameters)) (*function*), 37
NormEquation(K, y) (*function*), 85
NthPrime(n) (*function*), 27
NumberFieldSieve(n, F, m1, m2) (*function*), 40
NumberOfDivisors(f) (*function*), 35
NumberOfDivisors(n) (*function*), 22
NumberOfDivisors(Q) (*function*), 22
NumberOfGenerators(G) (*function*), 60
NumberOfPartitions(n) (*function*), 24
NumberOfPrimePolynomials(K, d) (*function*), 118
NumberOfPrimePolynomials(q, d) (*function*), 118

NumberOfPrimePolynomials(R, d) (*function*), 118
NumberOfRelationsRequired(P) (*function*), 43
NumberOfVariants(N) (*function*), 96
NumberOfVariants(q, v) (*function*), 96
Numerator and Denominator (*example*), 69
Numerator(q) (*function*), 69
Numerical Derivative (*example*), 184
Numerical Integration on the Unit Interval (*example*), 182
NumericalDerivative(f, n, z) (*function*), 184

O

OldCertificate(cert) (*function*), 26
One(F) (*function*), 78
One(P) (*function*), 107, 131
One(Q) (*function*), 66
One(R) (*function*), 7, 54, 152
One(Z) (*function*), 14
OptimalSkewness(F) (*function*), 47
Order(a) (*function*), 57, 86
Order(chi) (*function*), 60
Order(G) (*function*), 60
Order(x) (*function*), 100
OrderOfRootOfUnity(r, n) (*function*), 61

P

P ! s (*operation*), 107
P . 1 (*operation*), 107
P . i (*operation*), 131
P eq Q (*operation*), 109, 132
P ge Q (*operation*), 110
P gt Q (*operation*), 110
P le Q (*operation*), 110
P lt Q (*operation*), 110
P ne Q (*operation*), 110, 132
Parent(a) (*function*), 84, 98
Parent(F) (*function*), 80
Parent(n) (*function*), 18, 55
Parent(P) (*function*), 108, 131
Parent(p) (*function*), 110
Parent(Q) (*function*), 66
Parent(R) (*function*), 5, 53, 153
Parent(r) (*function*), 6, 68, 154
Parent(Z) (*function*), 15
Partial Factorization (*example*), 34
PartialFactorization(S) (*function*), 34
Partitions(n) (*function*), 24
Perfect Numbers (*example*), 33
Pi(R) (*function*), 159
pMinus1(n, B1) (*function*), 31
PolarToComplex(m, a) (*function*), 157
PollardRho(n) (*function*), 31
PollardRho(n, c, s, k) (*function*), 31
Polylog(m, f) (*function*), 167
Polylog(m, s) (*function*), 167
PolylogD(m, s) (*function*), 167
PolylogDold(m, s) (*function*), 167
PolylogP(m, s) (*function*), 167
Polynomial Decomposition (*example*), 115
Polynomial Factorization via LLL (*example*), 165
Polynomial Interpolation (*example*), 140
Polynomial(C, M) (*function*), 137
Polynomial(Q) (*function*), 107
Polynomial(R, f) (*function*), 107
Polynomial(R, Q) (*function*), 107
PolynomialAlgebra(R) (*function*), 105
PolynomialAlgebra(R, n) (*function*), 128
PolynomialAlgebra(R, n, order) (*function*), 129
PolynomialRing(R) (*function*), 105
PolynomialRing(R, n) (*function*), 128
PolynomialRing(R, n, order) (*function*), 129
PolynomialSieve(F, m, J0, J1, MaxAlpha) (*function*), 48
PositiveSum(m, i) (*function*), 180
PowerIdeal(R) (*function*), 10
PowerPolynomial(f,n) (*function*), 116
PowerRelation(r, k: parameters) (*function*), 164
pPlus1(n, B1) (*function*), 32
Precision(c) (*function*), 158
Precision(L) (*function*), 158
Precision(R) (*function*), 154
Precision(r) (*function*), 158
PreimageRing(Q) (*function*), 125
PreviousPrime(n) (*function*), 27
PreviousPrime(n: parameter) (*function*), 27
PrimalityCertificate(n) (*function*), 25
PrimeBasis(n) (*function*), 28, 33
PrimeDivisors(f) (*function*), 35
PrimeDivisors(n) (*function*), 28, 33
PrimeField(F) (*function*), 5, 76, 80
PrimeField(N) (*function*), 99
PrimeField(Q) (*function*), 66
PrimeField(R) (*function*), 153
PrimePolynomials(R, d) (*function*), 118
PrimePolynomials(R, d, n) (*function*), 118
PrimeRing(F) (*function*), 80
PrimeRing(P) (*function*), 108, 131
PrimeRing(R) (*function*), 5, 53
PrimeRing(Z) (*function*), 15
Primes Congruent to 3 Modulo 4 (*example*), 18
PrimesInInterval(b, e) (*function*), 27
PrimesUpTo(B) (*function*), 27
PrimitiveElement(F) (*function*), 79
PrimitiveElement(R) (*function*), 57
PrimitivePart(f) (*function*), 143
PrimitivePart(p) (*function*), 118
PrimitivePolynomial(F, m) (*function*), 87
PrimitiveRoot(m) (*function*), 36
PrimitiveRoot(R) (*function*), 57
Printing in Hexadecimal (*example*), 14
Probable Repunit Primes (*example*), 26

Projective Plane of a Nearfield (*example*), 101
ProjectivePlane(N : parameters) (*function*), 101
PseudoRemainder(f, g) (*function*), 116
Psi(s) (*function*), 177

Q

Q ! a (*operation*), 66
Q ! [a, b] (*operation*), 66
Q ! [a] (*operation*), 66
Q eq R (*operation*), 68
Q ne R (*operation*), 68
q-Expansion of a Weber Function (*example*), 175
q-Expansion of an Eisenstein Series (*example*), 172
QMatrix(f) (*function*), 122
Qround(q, M) (*function*), 71
quo< R | a_r, ..., a_r > (*constructor*), 9
quo< R | a_1, ..., a_r > (*constructor*), 123
quo< R | I > (*constructor*), 123
quo<Z | I> (*constructor*), 52
quo<Z | m> (*constructor*), 52
Quotrem(f, g) (*function*), 115
Quotrem(m, n) (*function*), 20

R

R ! a (*operation*), 7, 152
R ! k (*operation*), 54
R ! s (*operation*), 131
r * s (*operation*), 156
r *:= s (*operation*), 156
r + s (*operation*), 156
r +:= s (*operation*), 156
R / I (*operation*), 9
r / s (*operation*), 156
r /:= s (*operation*), 156
r - s (*operation*), 156
r -:= s (*operation*), 156
r ^ k (*operation*), 156
r ^:= s (*operation*), 156
R eq R (*operation*), 54
R eq S (*operation*), 6, 8, 154
R ne R (*operation*), 54
R ne S (*operation*), 6, 8, 154
RamificationIndex(I) (*function*), 51
RamificationIndex(I, p) (*function*), 51
Random(a, b) (*function*), 20
Random(b) (*function*), 20
Random(F) (*function*), 78
Random(G) (*function*), 59
Random(N) (*function*), 99
Random(Q, m) (*function*), 66
Random(R) (*function*), 7, 54
RandomBits(n) (*function*), 20
RandomConsecutiveBits(n, a, b) (*function*), 21
RandomExtension(F, n) (*function*), 75
RandomIrreduciblePolynomial(F, n) (*function*), 87
RandomPrime(n, a, b, x: parameter) (*function*), 21, 28
RandomPrime(n: parameter) (*function*), 20, 28
RandomPrimePolynomial(R, d) (*function*), 118
Rank(P) (*function*), 109, 132
RationalField() (*function*), 65
RationalReconstruction(s) (*function*), 71
Rationals() (*function*), 65
Re(c) (*function*), 157
Real Numbers and Loss of Precision (*example*), 148
Real(c) (*function*), 157
RealField() (*function*), 150
RealField(p) (*function*), 150
ReciprocalPolynomial(f) (*function*), 116
Reductum(f) (*function*), 116, 141
Reductum(f, i) (*function*), 141
Reductum(f, v) (*function*), 141
RemoveFiles(P) (*function*), 44
Rep(R) (*function*), 7
Representative(F) (*function*), 78
Representative(P) (*function*), 107, 131
Representative(Q) (*function*), 66
Representative(R) (*function*), 7, 54, 152, 153
Representative(Z) (*function*), 14
ResidueClassField(I) (*function*), 11
ResidueClassField(p) (*function*), 52
ResidueClassRing(m) (*function*), 52
ResidueClassRing(Q) (*function*), 52
RestrictedPartitions(n, k, M) (*function*), 24
RestrictedPartitions(n, M) (*function*), 24
Resultant(f, g) (*function*), 122
Resultant(f, g, i) (*function*), 146
Resultant(f, g, v) (*function*), 146
RingOfIntegers() (*function*), 13
RingOfIntegers(m) (*function*), 52
RingOfIntegers(Q) (*function*), 13, 65
RombergQuadrature(f, a, b: parameters) (*function*), 184
Root(a, n) (*function*), 86
Root(r, n) (*function*), 159
RootOfUnity(n, K) (*function*), 82
RootOfUnity(n, Q) (*function*), 66
Roots and Traces of Elements (*example*), 86
Roots in a Splitting Field (*example*), 83
Roots of a Polynomial with a Multiple Root (*example*), 162
Roots with Error Estimates (*example*), 163
Roots(f) (*function*), 82
Roots(p) (*function*), 113, 162
Roots(p, S) (*function*), 113
RootsInSplittingField(f) (*function*), 82
RootsNonExact(p) (*function*), 163
Round(n) (*function*), 20
Round(p) (*function*), 112
Round(q) (*function*), 71

Round(r) (*function*), 158
Round(x) (*function*), 39

S

s * t (*operation*), 35
s + t (*operation*), 35
s / t (*operation*), 35
s - t (*operation*), 35
s ^ k (*operation*), 35
Sec(c) (*function*), 168
Sec(f) (*function*), 168
Sec(r) (*function*), 168
Sech(r) (*function*), 170
Seqelt(s, F) (*function*), 79
SeqFact(s) (*function*), 34
Seqint(s, b) (*function*), 14
SequenceToElement(s, F) (*function*), 79
SequenceToFactorization(s) (*function*), 34
SequenceToInteger(s, b) (*function*), 14
Set(F) (*function*), 80
Set(R) (*function*), 53
SetDefaultRealField(R) (*procedure*), 150
SetPowerPrinting(F, l) (*procedure*), 77
SetPrimitiveElement(F, x) (*procedure*), 79
SetVerbose("Cunningham", b) (*procedure*), 30
SetVerbose("ECM", b) (*procedure*), 30
SetVerbose("ECPP", v) (*procedure*), 25
SetVerbose("Factorization", v) (*procedure*), 29
SetVerbose("FFLog", v) (*procedure*), 89
SetVerbose("MPQS", b) (*procedure*), 30
SetVerbose("NFS", v) (*procedure*), 40
SetVerbose("PolyFact", v) (*procedure*), 120, 144
SetVerbose("SmallRoots", v) (*procedure*), 114
ShiftLeft(n, b) (*function*), 17
ShiftRight(n, b) (*function*), 17
Sieve(K) (*procedure*), 89
Sign(f) (*function*), 147
Sign(n) (*function*), 20
Sign(p) (*function*), 118
Sign(q) (*function*), 70
Sign(r) (*function*), 159
Sign(x) (*function*), 39
Signature(Q) (*function*), 68
Signature(Z) (*function*), 15
Simple Nearfield Arithmetic (*example*), 98
SimpsonQuadrature(f, a, b, n) (*function*), 184
Sin(c) (*function*), 167
Sin(f) (*function*), 167
Sin(r) (*function*), 167
Sincos(f) (*function*), 168
Sincos(s) (*function*), 168
Sinh(f) (*function*), 170
Sinh(r) (*function*), 170
Sinh(s) (*function*), 170
SmallRoots(p, N, X) (*function*), 113
Solution(a, b) (*function*), 56
Solution(a, b, m) (*function*), 36
Solution(A, B, N) (*function*), 37
SplittingField(P) (*function*), 75
SplittingField(S) (*function*), 75
Sqrt(a) (*function*), 56, 86
Sqrt(c) (*function*), 159
Sqrt(r) (*function*), 159
Sqrt(x) (*function*), 62
Square Roots Modulo 2340 (*example*), 56
SquarefreeFactorization(f) (*function*), 35, 122, 143
SquarefreeFactorization(n) (*function*), 20
SquarefreePart(f) (*function*), 144
SquareRoot(a) (*function*), 56, 86
SquareRoot(c) (*function*), 159
SquareRoot(r) (*function*), 159
SQUFOF(n) (*function*), 32
SQUFOF(n, k) (*function*), 32
StirlingFirst(n, k) (*function*), 24
StirlingSecond(n, k) (*function*), 24
StoreFactor(n) (*procedure*), 30
StoreFactor(S) (*procedure*), 30
sub< R | n > (*constructor*), 53
sub< Z | n > (*constructor*), 15
sub<F | d> (*constructor*), 75
sub<F | f> (*constructor*), 75
SumNorm(f) (*function*), 147
SumNorm(p) (*function*), 118
SumOfDivisors(f) (*function*), 35
SumOfDivisors(n) (*function*), 22
SumOfDivisors(Q) (*function*), 22
Support(p) (*function*), 112
SwinnertonDyerPolynomial(n) (*function*), 126
SylvesterMatrix(f, g) (*function*), 122
Symmetric Bilinear Form of a Quadratic Polynomial (*example*), 141
SymmetricBilinearForm(f) (*function*), 141

T

Tan(c) (*function*), 168
Tan(f) (*function*), 168
Tan(r) (*function*), 168
Tanh(f) (*function*), 170
Tanh(r) (*function*), 170
TanhSinhIntegrationPoints(N,h) (*function*), 182
Term(f, i, k) (*function*), 136
Term(f, v, k) (*function*), 136
Terms(f) (*function*), 136
Terms(f, i) (*function*), 136
Terms(f, v) (*function*), 136
Terms(p) (*function*), 112
Theta(char, z, A) (*function*), 176
Theta(char, z, tau) (*function*), 176
TotalDegree(f) (*function*), 138
Trace(a) (*function*), 85
Trace(a, E) (*function*), 85
Trace(n) (*function*), 19

Trace(q) (*function*), 70
TraceAbs(a) (*function*), 85
TrailingCoefficient(f) (*function*), 135
TrailingCoefficient(f, i) (*function*), 135
TrailingCoefficient(f, v) (*function*), 135
TrailingCoefficient(p) (*function*), 112
TrailingTerm(f) (*function*), 136
TrailingTerm(f, i) (*function*), 136
TrailingTerm(f, v) (*function*), 136
TrailingTerm(p) (*function*), 112
TrapezoidalQuadrature(f, a, b, n) (*function*), 184
TrialDivision(n) (*function*), 31
TrialDivision(n, B) (*function*), 31
Truncate(n) (*function*), 20
Truncate(q) (*function*), 71
Truncate(r) (*function*), 158
TwoElementNormal(I) (*function*), 51
Type(R) (*function*), 5
Type(r) (*function*), 6

U

Unit Group of a Residue Class Ring (*example*), 58
Unit Group of a Subnearfield (*example*), 100
UnitGenerators(G) (*function*), 60
UnitGroup(F) (*function*), 80
UnitGroup(GrpPC, N) (*function*), 100
UnitGroup(GrpPerm, N) (*function*), 100
UnitGroup(N) (*function*), 100
UnitGroup(Q) (*function*), 67
UnitGroup(R) (*function*), 53, 57
UnitGroup(Z) (*function*), 15
UnivariatePolynomial(f) (*function*), 138

V

Valuation(f, g) (*function*), 116
Valuation(p) (*function*), 112
Valuation(x, I) (*function*), 51, 72
Valuation(x, p) (*function*), 20, 72
ValueList(chi) (*function*), 61
ValuesOnUnitGenerators(chi) (*function*), 61
VariantRepresentatives(q, v) (*function*), 96
Variants of a Dickson Pair (*example*), 96
VectorSpace(F, E) (*function*), 80
VectorSpace(F, E, B) (*function*), 80
VectorSpace(Q, Q) (*function*), 67

W

Ways to Define a Finite Field (*example*), 76
WeberF(s) (*function*), 175
WeberF1(s) (*function*), 175
WeberF2(g) (*function*), 175
WeberF2(s) (*function*), 175
WeierstrassSeries(z, F) (*function*), 173
WeierstrassSeries(z, L) (*function*), 173
WeierstrassSeries(z, q) (*function*), 173
WeierstrassSeries(z, t) (*function*), 173

X

x * y (*operation*), 38, 61
x + y (*operation*), 38
x / y (*operation*), 38, 61
x - y (*operation*), 38
x ^ n (*operation*), 38, 61
x ^ phi (*operation*), 61
x eq y (*operation*), 38, 60
x ge y (*operation*), 39
x gt y (*operation*), 38
x le y (*operation*), 38
x lt y (*operation*), 38
x ne y (*operation*), 38
XGCD(f, g) (*function*), 117
Xgcd(f, g) (*function*), 117
XGCD(m, n) (*function*), 21
Xgcd(m, n) (*function*), 21
XGCD(s) (*function*), 22
Xgcd(s) (*function*), 22

Z

Z ! a (*operation*), 13
Z ! [a] (*operation*), 13
Z eq R (*operation*), 16
Z ne R (*operation*), 16
Zassenhaus Nearfields (*example*), 97
ZassenhausNearfield(n) (*function*), 97
ZechLog(K, n) (*function*), 89
Zero(F) (*function*), 78
Zero(N) (*function*), 99
Zero(P) (*function*), 107, 131
Zero(Q) (*function*), 66
Zero(R) (*function*), 7, 54, 152
Zero(Z) (*function*), 14
ZetaFunction(R, n) (*function*), 180
ZetaFunction(s) (*function*), 180