HANDBOOK OF MAGMA FUNCTIONS

Volume 2

Basic Rings and Linear Algebra

John Cannon   Wieb Bosma

Claus Fieker   Allan Steel

Editors

Version 2.22

Sydney

June 9, 2016

ii

MAGMA
COMPUTER●ALGEBRA

HANDBOOK OF MAGMA FUNCTIONS

Editors:

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

Handbook Contributors:

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

Production Editors:

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

HTML Production:

Claus Fieker   *Allan Steel*

VOLUME 2: OVERVIEW

VOLUME 2: CONTENTS

PART III
BASIC RINGS

17 INTRODUCTION TO RINGS

Chapter 17

INTRODUCTION TO RINGS

17.1 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	Ch
$\mathbf{Z}$	ring of integers	RngInt	18
$\mathbf{Z}/m\mathbf{Z}$	ring of residue classes	RngIntRes	18
$R[x]$	univariate polynomial ring	RngUPol	23
$F[x]/f(x)$	polynomial factor ring	RngUPolRes	23
$R[x_1, \cdots, x_m]$	multivariate polynomial ring	RngMPol	24
$R[x_1, \cdots, x_m]^G$	invariant ring	RngInvar	116
$R[[x]]$	power series ring	RngSer	49
O	order in a number field	RngOrd	38
O	order in a function field	RngFunOrd	44
$\mathbf{Z}_p$	p-adic ring	RngPad	47
R_m	local ring	RngLoc	47
V	valuation ring	RngVal	52

Table 1: The main types of Ring in MAGMA.

symbol	description	Category	Ch
$\mathbf{Q}$	rational field	FldRat	20
$\mathbf{F}_q$	finite field	FldFin	21
$F(x_1, \cdots, x_m)$	rational function field	FldFunRat	43
$F((x))$	field of Laurent series	RngSerLaur	49
$\mathbf{Q}(\sqrt{D})$	quadratic number field	FldQuad	36
$\mathbf{Q}(\zeta_n)$	cyclotomic number field	FldCyc	37
$\mathbf{Q}(\alpha)$	number field	FldNum	35
$F(x)(\alpha)$	function field	FldFun	44
$\mathbf{Q}_p$	p-adic field	FldPad	47
$\mathbf{Q}_p(\alpha)$	local field	FldLoc	47
$\mathbf{R}$	real field	FldRe	25
$\mathbf{C}$	complex field	FldCom	25

Table 2: The main types of Field in MAGMA.

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 **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.

17.2 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.

17.2.1 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 **Z** by repeated application of a handful of fundamental mathematical constructions. The first such construction is *forming fractions*: the rational field **Q** can be obtained as the field of fractions of **Z**. The second construction is that of forming quotients: in this way the rings $\mathbf{Z}/m\mathbf{Z}$ are obtained from **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 $\mathbf{Z}/p\mathbf{Z}$ 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 $K = \mathbf{Q}(\alpha)$ is to regard it as a quotient of the polynomial ring $\mathbf{Q}[X]$ and the ideal generated by the minimal polynomial f of the primitive element α:

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

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)`.

17.3   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 **Q** can be coerced into **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)`.

17.3.1 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 **Q**, that is, we clearly wish to have automatic coercion from **Z** to **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 $\mathbf{Z} \to R$, hence any integer can be coerced automatically into any ring.

Table 3 gives a summary for all cases in which MAGMA will apply automatic coercion: if a ring operation is attempted on an element from one of the structures in the first row, and the specifications for the operation require that the argument is an element from a structure in the first column, the element will be coerced into the structure indicated by the table.

Automatic Coercion (Ring Elements)										
	$\mathbf{F}_{p^s}$	$\mathbf{Z}/n\mathbf{Z}$	$\mathbf{Z}$	$\mathbf{Q}$	$\mathbf{Q}(\sqrt{\Delta_2})$	$\mathbf{Q}(\zeta_n)$	L	O_L	$\mathbf{R}_n$	$\mathbf{C}_n$
$\mathbf{F}_{p^r}$	$\subset$	$-$	$\mathbf{F}_{p^r}$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
$\mathbf{Z}/m\mathbf{Z}$	$-$	$=$	$\mathbf{Z}/m\mathbf{Z}$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
$\mathbf{Z}$	$\mathbf{F}_{p^s}$	$\mathbf{Z}/n\mathbf{Z}$	$\mathbf{Z}$	$\mathbf{Q}$	$\mathbf{Q}(\sqrt{\Delta_2})$	$\mathbf{Q}(\zeta_n)$	L	O_L	$\mathbf{R}_n$	$\mathbf{C}_n$
$\mathbf{Q}$	$-$	$-$	$\mathbf{Q}$	$\mathbf{Q}$	$\mathbf{Q}(\sqrt{\Delta_2})$	$\mathbf{Q}(\zeta_n)$	L	$-$	$\mathbf{R}_n$	$\mathbf{C}_n$
$\mathbf{Q}(\sqrt{\Delta_1})$	$-$	$-$	$\mathbf{Q}(\sqrt{\Delta_1})$	$\mathbf{Q}(\sqrt{\Delta_1})$	$=$	$-$	$-$	$-$	$-$	$-$
$\mathbf{Q}(\zeta_m)$	$-$	$-$	$\mathbf{Q}(\zeta_m)$	$\mathbf{Q}(\zeta_m)$	$-$	$\mathbf{Q}(\zeta_{\mathrm{lcm}(m,n)})$	$-$	$-$	$-$	$-$
K	$-$	$-$	K	K	$-$	$-$	$=$	$K=L$	$-$	$-$
O_K	$-$	$-$	O_K	$-$	$-$	$-$	$K=L$	$=$	$-$	$-$
$\mathbf{Q}_p$	$-$	$-$	$\mathbf{Q}_p$	$\mathbf{Q}_p$	$-$	$-$	$-$	$-$	$-$	$-$
$\mathbf{Z}_p$	$-$	$-$	$\mathbf{Z}_p-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
$\mathbf{R}_m$	$-$	$-$	$\mathbf{R}_m$	$\mathbf{R}_m$	$-$	$-$	$-$	$-$	$\mathbf{R}_{\max(m,n)}$	$\mathbf{C}_{\max(m,n)}$
$\mathbf{C}_m$	$-$	$-$	$\mathbf{C}_m$	$\mathbf{C}_m$	$-$	$-$	$-$	$-$	$\mathbf{C}_{\max(m,n)}$	$\mathbf{C}_{\max(m,n)}$

Table 3.

The symbols in the table have the following meaning:

$-$ indicates that automatic coercion will *not* take place; as the table shows for instance, automatic coercion will not take place when we try to add a finite field element and an element of $\mathbf{Z}/n\mathbf{Z}$ (not even when n is prime and of the same characteristic as the field).

$\subset$ indicates that automatic coercion will *only* take place if one structure is contained in the other, or MAGMA can find a common overstructure to both structures. Thus the top-left entry in the table indicates that two finite field elements can be added if they are members of finite fields F_1 and F_2 such that $F_1 \subset F_2$ or $F_2 \subset F_1$, or both fields have been created inside a field F, so $F_1 \subset F \supset F_2$.

$=$ The $=$ is used to denote that automatic coercion only takes place if both structures are the same. The entry $K = L$ is used to indicate that an element of an order O_K will only be coerced into the number field L if $K = L$.

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 $\mathbf{Z}[X]$ and g of $\mathbf{Z}[X,Y]$, 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 $M_{n,n}(\mathbf{Z})$ and $M_{n,n}(\mathbf{Q})$, nor elements from $M_{2,2}(\mathbf{Z})$ and $M_{3,3}(\mathbf{Z})$.

17.3.2 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 $\mathbf{Z}/p\mathbf{Z}$, 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 $\mathbf{Z}/p\mathbf{Z}$ can be cast into an element of F using
`F ! x;`
Table 4 describes the possibilities for using `!` for coercion in rings. It shows when it is possible to coerce an element from a structure in the first row into a structure in the first column.

! **Non-Automatic Coercion (Ring Elements)**										
↙	$\mathbf{F}_{p^s}$	$\mathbf{Z}/n\mathbf{Z}$	$\mathbf{Z}$	$\mathbf{Q}$	$\mathbf{Q}(\sqrt{\Delta_2})$	$\mathbf{Q}(\zeta_n)$	L	O_L	$\mathbf{R}_n$	$\mathbf{C}_n$
$\mathbf{F}_{p^r}$	$s\|r$ or $\ni$	$n=p$	+	−	−	−	−	−	−	−
$\mathbf{Z}/m\mathbf{Z}$	$m=p, s=1$	$m\|n$	+	−	−	−	−	−	−	−
$\mathbf{Z}$	$s=1$ or $\ni$	+	+	$\ni$	$\ni$	$\ni$	$\ni$	$\ni$	−	−
$\mathbf{Q}$	−	−	+	+	$\ni$	$\ni$	$\ni$	$\ni$	−	−
$\mathbf{Q}(\sqrt{\Delta_1})$	−	−	+	+	$\Delta_1=\Delta_2$ or $\ni$	$\ni$	−	−	−	−
$\mathbf{Q}(\zeta_m)$	−	−	+	+	$\supset$ or $\ni$	$n\|m$ or $\ni$	−	−	−	−
K	−	−	+	+	−	−	$K=L$	$K=L$	−	−
O_K	−	−	+	$\ni$	−	−	$K=L,\ni$	$K=L$	−	−
$\mathbf{Q}_p$	$s=1$	$n=p$	+	$\ni$	−	−	−	−	−	−
$\mathbf{Z}_p$	$s=1$	$n=p$	+	−	−	−	−	−	−	−
$\mathbf{R}_m$	−	−	+	+	$\Delta_2>0$ or $\ni$	$\ni$	−	−	+	$\ni$
$\mathbf{C}_m$	−	−	+	+	+	+	−	−	+	+

Table 4.

The symbols are the same as those in Table 3 except:

+   indicates that coercion can take place without restrictions (sometimes it will be done automatically if necessary);

$|, =$ (In)equalities on parameters of the structures indicate the restrictions for `!` to work; thus, an element from $\mathbf{F}_{p^s}$ can only be coerced into $\mathbf{Z}/m\mathbf{Z}$ if $s=1$ and $m=p$;

∋   The ∋ symbols in this table indicates that coercion only applies to certain elements of the domain; thus only those elements of $\mathbf{Q}$ can be coerced into $\mathbf{Z}$ that are in fact integers.

or   In some cases coercion may either take place if some condition on parameters is satisfied *or* on a subset of the domain; thus the entry $s|r$ or $\ni$ for coercion of $\mathbf{F}_{p^s}$ into $\mathbf{F}_{p^r}$ indicates that such coercion is always possible if s divides r, and only on a subset of $\mathbf{F}_{p^s}$ (like $\mathbf{F}_p$) in general (note that the characteristics have to be the same).

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 $1 \le k \le n$, 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.

17.4 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.

17.4.1 Related Structures

Parent(R)

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

Category(R)

Type(R)

   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)

   For a field F, this returns either $\mathbf{F}_p$, if the characteristic p of F is positive, or $\mathbf{Q}$, 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)

   For a unitary ring R, this returns either $\mathbf{Z}/n\mathbf{Z}$, if the characteristic n of R is positive, or $\mathbf{Z}$, 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)

Center(R)

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

17.4.2 Numerical Invariants

Characteristic(R)

   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

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

17.4.3   Predicates and Boolean Operations

IsCommutative(R)

> 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)

> 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)

> 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)

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

IsField(R)

> 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)

> 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)

> 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)

> 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)

> 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)

IsPrincipalIdealDomain(R)

> 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)**

> **IsPrincipalIdealRing(R)**

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)**

> **IsUniqueFactorizationDomain(R)**

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)**

> **IsIntegralDomain(R)**

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)**

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

> **R eq S**

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**

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.

17.5 Generic Element Functions

17.5.1 Parent and Category

> **Parent(r)**

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)**

> **Type(r)**

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.

17.5.2 Creation of Elements

`Zero(R)`

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

`One(R)`

`Id(R)`

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

`R ! a`

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)`

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

`Representative(R)`

`Rep(R)`

A representative element of the finite ring R.

17.5.3 Arithmetic Operations

`+a`

Element a.

`-a`

The negation (additive inverse) of element a.

`a + b`

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`

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`

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`

Form the k-th power of the ring element a, for small, non-negative, k. If $a = 0$ then we must have $k > 0$.

`a ^ -k`

Form the k-th power of the multiplicative inverse of the unit a.

`a / b`

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`

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

`a -:= b`

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

`a *:= b`

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

`a /:= b`

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

`a ^:= k`

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

17.5.4 Equality and Membership

`a eq b`

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

`a ne b`

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

`R eq S`

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

`R ne S`

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

`a in R`

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

`a notin R`

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

17.5.5   Predicates on Ring Elements

`IsZero(a)`

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

`IsOne(a)`

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

`IsMinusOne(a)`

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

`IsUnit(a)`

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

`IsIdempotent(x)`

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

`IsNilpotent(x)`

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

`IsZeroDivisor(x)`

> 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)`

> 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)`

> 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.

17.5.6 Comparison of Ring Elements

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

---

a gt b

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

---

a ge b

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

---

a lt b

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

---

a le b

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

Maximum(a, b)

> 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)

> The maximum of the sequence Q of ring elements.

Minimum(a, b)

> 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)

> The minimum of the sequence Q of ring elements.

17.6   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.

17.6.1   Defining Ideals and Quotient Rings

---

ideal< R | a_1, ..., a_r >

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 | a_r, ..., a_r >

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

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, $\mathbf{Z}/I$ is computing the (fractional) ideal J such that $JI = \mathbf{Z}$. The technical reason for this is that both $\mathbf{Z}$ 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)

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

17.6.2 Arithmetic Operations on Ideals

I + J

> 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

> 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 $a_1 b_1 + \cdots + a_n b_n$, with $a_i \in I$ and $b_j \in J$.

I meet J

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

17.6.3 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

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

---
a notin I

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

I eq J

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

I ne J

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

I subset J

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

I notsubset J

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

17.7   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.

17.7.1   Residue Class Fields

ResidueClassField(I)

> 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.

17.7.2   Localization

loc< R | a$_1$, ..., a$_r$ >

> 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)

> 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.

17.7.3   Completion

comp< R | a$_1$, ..., a$_r$ >

> 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)

> 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.

17.7.4   Transcendental Extension

ext< R | >

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

ext< R, n | >

> 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).

18 RING OF INTEGERS

Chapter 18

RING OF INTEGERS

18.1 Introduction

This Chapter describes the operators and functions for working with the ring of rational integers $\mathbf{Z}$.

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.

18.1.1 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).

18.1.2 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 17.

18.1.3 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 |  >
```

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

Example H18E1

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

18.2   Creation Functions

18.2.1   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 **Z** are always ideals; see the section on ideals for details.

> `IntegerRing()`
>
> `Integers()`
>
> `RingOfIntegers()`
>
> `IntegerRing(Q)`
>
> `Integers(Q)`
>
> `RingOfIntegers(Q)`

Create the ring of integers **Z**.

18.2.2   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 both decimal and hexadecimal notation.

> `a₁a₂...aᵣ`

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

> `0xa₁a₂...aᵣ`

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

> `elt<  Z | a₁a₂...aᵣ  >`

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

elt< Z	0x$a_1a_2\ldots a_r$ >

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

Z ! a

Z ! [a]

> 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 H18E2_____

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

_____

One(Z)		Identity(Z)

Zero(Z)		Representative(Z)

> These generic functions (cf. Chapter 17) create 1, 1, 0, and 0 respectively, in the integer ring **Z**.

18.2.3   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 H18E3_____

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

18.2.4   Element Conversions

FactorizationToInteger(s)

FactorisationToInteger(s)

Facint(s)

> 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)

Intseq(n, b)

> 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)

Seqint(s, b)

> 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)

> 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)

> 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)

> The sequence $[n]$ which can be coerced back into **Z**.

Denominator(n)

> The denominator of n, ie. 1.

18.3   Structure Operations

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

18.3.1   Related Structures

Category(Z)   **Parent(Z)**   **PrimeRing(Z)**   **Center(Z)**

AdditiveGroup(Z)

> 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)

UnitGroup(Z)

> 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)

> The class group of the ring of **Z** (which is trivial).

FieldOfFractions(Z)

> Create the field of fractions **Q** of the ring of rational integers.

sub< Z | n >

> 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.

18.3.2   Numerical Invariants

```
Characteristic(Z)
```

```
Signature(Z)
```
The signature of **Z** as an order of **Q**, i.e. `1, 0`.

18.3.3   Ring Predicates and Booleans

```
IsCommutative(Z)          IsUnitary(Z)
```
```
IsFinite(Z)        IsOrdered(Z)
```
```
IsField(Z)         IsEuclideanDomain(Z)
```
```
IsPID(Z)      IsUFD(Z)
```
```
IsDivisionRing(Z)          IsEuclideanRing(Z)
```
```
IsPrincipalIdealRing(Z)        IsDomain(Z)
```
```
Z eq R          Z ne R
```

18.4   Element Operations

18.4.1   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		– n							
m + n		m – n		m * n		n ^ k		m / n	
m +:= n		m –:= n		m *:= n		m /:= n		m ^:= k	

n div m

> 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

> 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)

> 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		n mod:= m

18.4.2 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)

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

ShiftRight(n, b)

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

ModByPowerOf2(n, b)

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

18.4.3 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)

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

BitwiseAnd(x, y)

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

BitwiseOr(x, y)

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

BitwiseXor(x, y)

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

18.4.4 Equality and Membership

m eq n   m ne n

n in R   n notin R

18.4.5 Parent and Category

Parent(n)   Category(n)

18.4.6 Predicates on Ring Elements

IsEven(n)

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

IsOdd(n)

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

IsDivisibleBy(n, d)

> 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)

> 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)

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

IsPower(n)

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)

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)

   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 the section 18.9 for a complete description of this function.

Example H18E4_____

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)

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

IsSinglePrecision(n)

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)   IsOne(n)   IsMinusOne(n)

IsNilpotent(n)   IsIdempotent(n)

IsUnit(n)   IsZeroDivisor(n)   IsRegular(n)

IsIrreducible(n)   IsPrime(n)

18.4.7   Comparison of Ring Elements

m gt n		m ge n		m lt n		m le n

Maximum(m, n)		Maximum(Q)
Minimum(m, n)		Minimum(Q)

18.4.8   Conjugates, Norm and Trace

ComplexConjugate(n)

> The complex conjugate of n, which will be the integer n itself.

Conjugate(n)

> The conjugate of n, which will be the integer n itself.

Norm(n)

> The norm in $\mathbf{Q}$ of n, which will be the integer n itself.

EuclideanNorm(n)

> The Euclidean norm (length) of n, which will equal the absolute value of n.

Trace(n)

> The trace (in $\mathbf{Q}$) of n, which will be the integer n itself.

MinimalPolynomial(n)

> 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.

18.4.9   Other Elementary Functions

AbsoluteValue(n)

Abs(n)

> Absolute value of the integer n.

Ilog2(n)

> The integral part of the logarithm to the base two of the positive integer n.

Ilog(b, n)

> 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)`

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 $0 \le r < n$ if $n > 0$ and $n < r \le 0$ if $n < 0$.

`Valuation(x, p)`

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)`

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)`

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

`Ceiling(n)`

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

`Floor(n)`

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

`Round(n)`

This function rounds the integer n to itself.

`Truncate(n)`

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

`SquarefreeFactorization(n)`

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)`

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.

18.5 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)

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

Random(b)

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)

A random integer m such that $0 \le m < 2^n$, 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*)

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 $m > 34 \cdot 10^{13}$ 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*)

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 $m > 34 \cdot 10^{13}$ 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)

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

18.6   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) |

| Gcd(m, n) |

| GCD(m, n) |

> 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) |

| Gcd(s) |

| GCD(s) |

> 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) |

| Xgcd(m, n) |

| XGCD(m, n) |

> 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) |

| Xgcd(s) |

| XGCD(s) |

> 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)

Lcm(m, n)

LCM(m, n)

> 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)

Lcm(s)

LCM(s)

> Least common multiple of the sequence of integers s.

18.7   Arithmetic Functions

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

CarmichaelLambda(n)

CarmichaelLambda(Q)

CarmichaelLambda(Q)

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

DickmanRho(u)

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

FactoredCarmichaelLambda(n)

FactoredCarmichaelLambda(Q)

FactoredCarmichaelLambda(Q)

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

DivisorSigma(i, n)

DivisorSigma(i, Q)

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

NumberOfDivisors(n)

NumberOfDivisors(Q)

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

SumOfDivisors(n)

SumOfDivisors(Q)

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

EulerPhi(n)

EulerPhi(Q)

EulerPhi(Q)

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

FactoredEulerPhi(n)

FactoredEulerPhi(Q)

FactoredEulerPhi(Q)

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

EulerPhiInverse(m)

EulerPhiInverse(Q)

> 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)

FactoredEulerPhiInverse(Q)

> 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)

> 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)

> 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)

> 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)

MoebiusMu(Q)

> 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 H18E5

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
```

18.8   Combinatorial Functions

| Binomial(n, r) |

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

| Multinomial(n, [r$_1$, ... r$_n$]) |

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

| Factorial(n) |

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

| IsFactorial(n) |

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

| Partitions(n) |

> 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) |

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

| RestrictedPartitions(n, Q) |

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

| RestrictedPartitions(n, k, M) |

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

| StirlingFirst(n, k) |

> The Stirling number of the first type, $s(n, k) = \begin{bmatrix} n \\ k \end{bmatrix}$, where n and k are non-negative integers.

| StirlingSecond(n, k) |

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

Bell(n)

> 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)

> 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)

> 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)

> 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.

18.9   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.

18.9.1   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

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

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 $2 \dots n - 1$ will witness the compositeness of n. Thus randomly choosing a will usually quickly expose compositeness. Unless more than 3/4 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 $34 \cdot 10^{13}$ for which a witness does not exist among $a = 2, 3, 5, 7, 11, 13, 17$ ([Jae93]). Using these values of a for candidate witnesses it is certain that for any number n less than $34 \cdot 10^{13}$ the test will either find a witness or correctly declare n prime.

    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, as implemented by François Morain (Ecole Polytechnique and INRIA). The ECPP program in turn uses the BigNum package developed jointly by INRIA and Digital PRL. 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]).

    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)

IsPrime(n: *parameter*)

    Proof   BOOLELT   *Default :* `true`

        Returns `true` iff the integer n is prime. A rigorous method will be used, unless $n > 34 \cdot 10^{13}$ and the 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)

        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)

IsPrimeCertificate(cert)

ShowCertificate	BOOLELT	*Default :* true
Trust	RNGINTELT	*Default :* 0

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 list with data in the format described in [AM93].

To verify that a given number is prime from its primality certificate, the function IsPrimeCertificate is used. By default, this outputs only the result of the verification : true or false. If the user wishes to see the stages of the verification, the parameter ShowCertificate should be set to true. This is rather verbose as it shows the verification of primality of all small factors that need to be shown to be prime at each substage of the algorithm. It is usually more convenient to set the parameter Trust to a positive integer N which means that asserted primes less than N are not checked. This slightly reduces the time for the verification, but more importantly, it greatly reduces the output of ShowCertificate.

IsProbablePrime(n: *parameter*)

IsProbablyPrime(n: *parameter*)

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 $n < 34 \cdot 10^{13}$, 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)

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 H18E6 _____

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);
>    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).

18.9.2   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)

NextPrime(n: *parameter*)

   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)

PreviousPrime(n: *parameter*)

   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)

This function lists the primes up to (and including) the (positive) bound B. The algorithm is not super-optimised, but is reasonable.

PrimesInInterval(b, e)

This function lists the primes in the interval from b to e, including the endpoints. The algorithm is not very optimised.

NthPrime(n)

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

RandomPrime(n: *parameter*)

   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 $m > 34 \cdot 10^{13}$ 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*)

   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 $m > 34 \cdot 10^{13}$ 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)

PrimeDivisors(n)

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

18.10   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.

18.10.1   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.

> **SetVerbose("Factorization", v)**

> (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)**

> **Factorisation(n)**

> **Factorization(n:** *parameters***)**

> **Factorisation(n:** *parameters***)**

> 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.

> | Proof | BoolElt | *Default :* `true` |
> |---|---|---|
> | Bases | RngIntElt | *Default :* 20 |

> 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).

> | TrialDivisionLimit | RngIntElt | *Default :* 10000 |
> |---|---|---|

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

> SQUFOFLimit   RNGINTELT   *Default :* 24

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`).

> PollardRhoLimit   RNGINTELT   *Default :* 8191

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

> ECMLimit   RNGINTELT   *Default :*

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.

> MPQSLimit   RNGINTELT   *Default :* ∞

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 `Verbose("Factorization", true)`.

18.10.2   Storing Potential Factors

As of 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)

> StoreFactor(S)

> (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()

>   Return a sequence containing the currently stored integers.

ClearStoredFactors()

>   Clear the list of stored integer factors.

18.10.3   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)

SetVerbose("ECM", b)

SetVerbose("MPQS", b)

>   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)

>   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)

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)

TrialDivision(n, B)

Proof	BOOLELT	*Default :* `true`
Bases	RNGINTELT	*Default :* 20

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. The function returns a factorization sequence and a sequence containing an unfactored composite that remains.

PollardRho(n)

PollardRho(n, c, s, k)

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)

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)		

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)		

SQUFOF(n, k)		

| 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)		

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)

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)

MPQS(n, D)

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 name of a directory (which should not yet exist) may be specified as a string D where files used by MPQS will be stored. By default, the directory indicated by the environment variable MAGMA_QS_DIR will be used, and if that has not been set, the directory /tmp. It is possible to assist the master running the main MAGMA job by generating relations on other machines (slaves), starting an auxiliary process on such machine, in the directory D, by typing magma -q D machine where machine is the name of the machine. The function returns two values: a factorization sequence and a sequence containing unfactored composite factors.

18.10.4   Factorization Related Functions

ECMOrder(p, s)

ECMFactoredOrder(p, s)

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)

PrimeDivisors(n)

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

Divisors(n)

Divisors(f)

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)

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 H18E7_____

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

PartialFactorization(S)

Given a sequence of non-zero integers S, return, for each integer $S[i]$, two factorization lists F_i and G_i, such that $S[i] = \texttt{Facint}(F_i)*\texttt{Facint}(G_i)$. 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 H18E8_____

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> ]
  ]
]
```

18.11 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.

18.11.1   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)`
>
> `FactorizationToInteger(f)`

>> Create the integer corresponding to the factorization sequence f.

> `SeqFact(s)`
>
> `SequenceToFactorization(s)`

>> 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)`
>
> `ElementToSequence(f)`

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

18.11.2   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`   `s - t`   `s * t`   `s / t`   `s ^ k`

18.11.3   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)`   `Gcd(s, t)`   `SquarefreeFactorization(f)`

> `MoebiusMu(f)`   `Divisors(f)`   `PrimeDivisors(f)`

> `NumberOfDivisors(f)`   `SumOfDivisors(f)`

18.11.4   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)` | `IsOdd(s)` | `IsEven(s)` | `IsUnit(s)` |

| `IsPrime(s)` | `IsPrimePower(s)` | `IsSquare(s)` | `IsSquarefree(s)` |

18.12   Modular Arithmetic

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

18.12.1   Arithmetic Operations

`Modexp(n, k, m)`

> 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`

> 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)`
`InverseMod(n, m)`

> 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)`

> 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)`

> 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)`

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

PrimitiveRoot(m)

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

18.12.2 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 19.4 for details.

Solution(a, b, m)

> 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)

CRT(X, N)

> 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)

> 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)

NormEquation(d, m: *parameters*)

   Factorization   [<RNGINTELT, RNGINTELT>]

> 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 H18E9_____

```
> d := 9574400000957440000002277749760;
> m := 510219776051021977601213812848064;
> time NormEquation(d, m);
true 98 73
Time: 2.990
> time f := Factorization(m);
Time: 4.670
> f;
[ <2, 2>, <19, 1>, <6713418105934499705279129116421>, 1> ]
> time NormEquation(d, m: Factorization := f);
true 98 73
Time: 0.420
```

_____

18.13   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`.

18.13.1   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() |

      The positive infinity object.

| MinusInfinity() |

      The negative infinity object.

18.13.2 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

x + y		x - y		x * y		x / y		x ^ n

18.13.3 Comparison

Infinite objects may be compared with themselves and finite quantities.

x eq y		x ne y		x lt y		x le y		x gt y		x ge y

Maximum(x, y)		Minimum(x, y)

18.13.4 Miscellaneous

Sign(x)

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

Abs(x)
AbsoluteValue(x)

> Returns the positive infinite object.

Round(x)
Floor(x)
Ceiling(x)

> 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)

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

18.14   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.

18.14.1   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)

> 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.

18.14.2 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)

NFS(n, F, m1, m2)

> 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.

18.14.3 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)

> 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 H18E10_____

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.

_____

18.14.3.1   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 H18E11_____

Sample attributes for a 70-digit number:

```
> n := 5235869680233366295366904510725458053043111241035678897933802235060927;
> 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 H18E12_____

Sample attributes for an 80-digit number:

```
> n := 18718318663576864934511227229510402220632793503837386502539069334890072\
> 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 H18E13_____

Sample attributes for an 87-digit number:

```
> n := 121186187324634274722191791046317677651078393842196124697808418768214498\
> 2402918637227743;
> P<X,Y> := PolynomialRing(Integers(),2);
> F := 190512000*X^4 - 450872401242*X^3*Y +
>     1869594915648551*X^2*Y^2 + 2568544235742498*X*Y^3 -
>     93229655834198010101004*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.

18.14.3.2 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)**

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

> **FindRelations(P)**

> 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.

18.14.3.3 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)

> 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)

> 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)

> 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)

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

CreateCharacterFile(P, cc)

> 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.

18.14.3.4 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)

> Finds dependencies between relations in the NFS process P.

18.14.3.5 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)

> 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)

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

18.14.4   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)

> Deletes any data files created by the NFS process P.

MergeFiles(S, fn)

> 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.

18.14.4.1   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.

18.14.5 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 H18E14

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 := 70478782497479747987234958341;
> 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.

18.14.6   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)`

>   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)`

>   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`.

18.14.7 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)**

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)**

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)**

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 H18E15_____

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;
3596354707256625320407673937416777014871521894980389
> 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;
>   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 20.969934467920612180646408
> MurphyAlphaApproximation( F, 2000 );
-0.054271615763014144950015842
> time NFS( n, F, best_m, 1 );
...
```

BestTranslation(F, m, a)

> 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)

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 $G = F + j_1 x^2 y^{d-2} - (|j_0| + j_1 m) x y^{d-1} + (j_0 m) y^d$, 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.

18.15 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.

[**Jae93**] G. Jaeschke. On strong pseudoprimes to several bases. *Math. Comp.*, 61:915 – 926, 1993.

[**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*, pages 372–385, Berlin, 1995. Springer.

[**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.

[**Mar95**] G. Marsaglia. DIEHARD: a battery of tests of randomness. URL: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.

[**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.
URL:http://web.comlab.ox.ac.uk/oucl/work/richard.brent/ftp/Murphy-thesis.ps.gz.

[**MZ91**]　　George Marsaglia and Arif Zaman.　A new class of random number generators. *Ann. Appl. Probab.*, 1(3):462–480, 1991.

[**Sch71**]　　Arnold Schönhage.　Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.

[**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.

19 INTEGER RESIDUE CLASS RINGS

Chapter 19

INTEGER RESIDUE CLASS RINGS

19.1 Introduction

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

19.2 Ideals of Z

The theory of ideals of **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 **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 >

> Given the ring of integers **Z** and an integer a, return the ideal of **Z** generated by a.

Example H19E1_____

We construct some ideals of **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;
true
> I0 := ideal< Z | 0 >;
> 0 in I0;
true
> 1 in I0;
false
```

We check that **Z** is regarded as an ideal.

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

19.3   Z as a Number Field Order

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

`Decomposition(R, p)`

    Returns the ideal decomposition of the prime p, i.e. a list `[ < ideal<Z|p>, 1> ]` as in the number field case.

`Generator(I)`

    A generator for the given ideal.

`RamificationIndex(I, p)`

`RamificationIndex(I)`

    The ramification index of I over **Z** which is always 1.

`Degree(I)`

    The inertia degree of the ideal I, which is always 1.

`TwoElementNormal(I)`

    Two integers that generate the ideal I. In this case the generator is returned twice.

`ChineseRemainderTheorem(I, J, a, b)`

    The Chinese remainder theorem for ideals. Given ideals I and J of **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)`

    The valuation of the integer x at the prime ideal I.

`ClassRepresentative(I)`

    The representative of the ideal I of **Z** in the basis of the class group.

19.4   Residue Class Rings

The ring $\mathbf{Z}/m\mathbf{Z}$ 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 $\mathbf{Z}/m\mathbf{Z}$ 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 $\mathbf{Z}/m\mathbf{Z}$ 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 21 takes full advantage of the special structure of finite fields and leads to superior performance.

19.4.1   Creation

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

| quo< Z \| I > |

Given the ring of integers $\mathbf{Z}$, 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 $\mathbf{Z}$ as a number field order, returns the ideal quotient, namely $\mathbf{Z}$ itself (see Section 19.3 above, and the example in Sections 19.2).

| quo< Z \| m > |

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

| ResidueClassRing(m) |

| IntegerRing(m) |

| Integers(m) |

| RingOfIntegers(m) |

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

| ResidueClassField(p) |

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

> ResidueClassRing(Q)

> IntegerRing(Q)

> Integers(Q)

Create the residue class ring $\mathbf{Z}/m\mathbf{Z}$, 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 H19E2_____

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
```

19.4.2   Coercion

As can be seen from the tables in Chapter 17, automatic coercion takes place between $\mathbf{Z}/m\mathbf{Z}$ and $\mathbf{Z}$ so that a binary operation like $+$ applied to an element of $\mathbf{Z}/m\mathbf{Z}$ and an integer will result in a residue class from $\mathbf{Z}/m\mathbf{Z}$.

Using !, elements from a prime field $\mathbf{F}_p$ can be coerced into $\mathbf{Z}/p\mathbf{Z}$, and elements from $\mathbf{Z}/p\mathbf{Z}$ can be coerced into $\mathbf{F}_{p^r}$. Also, transitions between $\mathbf{Z}/m\mathbf{Z}$ and $\mathbf{Z}/n\mathbf{Z}$ 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 $\mathbf{Z}/m\mathbf{Z}$ is coerced into $\mathbf{Z}/n\mathbf{Z}$ with m dividing n – the result will be the residue class containing the representative for r.

Example H19E3_____

```
> 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.

19.4.3   Elementary Invariants

| Characteristic(R) | | # R |

| Modulus(R) |

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

| FactoredModulus(R) |

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

19.4.4   Structure Operations

| AdditiveGroup(R) |

> Given $R = \mathbf{Z}/m\mathbf{Z}$, 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 $\mathbf{Z}/m\mathbf{Z}$, sending $A.1$ to 1.

| MultiplicativeGroup(R) |
| UnitGroup(R) |

> Given $R = \mathbf{Z}/m\mathbf{Z}$, 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 > |

> 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) |

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

| Category(R) | | Parent(R) | | PrimeRing(R) |

| Center(R) |

19.4.5 Ring Predicates and Booleans

IsCommutative(R)	IsUnitary(R)

IsFinite(R)	IsOrdered(R)

IsField(R)	IsEuclideanDomain(R)

IsPID(R)	IsUFD(R)

IsDivisionRing(R)	IsEuclideanRing(R)

IsPrincipalIdealRing(R)	IsDomain(R)

R eq R	R ne R

19.4.6 Homomorphisms

Ring homomorphisms with domain $\mathbf{Z}/m\mathbf{Z}$ are completely determined by the image of 1. As usual (see Chapter 18), we require our homomorphisms to map 1 to 1. Therefore, the general homomorphism constructor with domain $\mathbf{Z}/m\mathbf{Z}$ needs no arguments.

hom< R -> S | >

> 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!

19.5 Elements of Residue Class Rings

19.5.1 Creation

elt< R | k >

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

R ! k

> 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 $\mathbf{F}_p$ in the case $R = \mathbf{Z}/p\mathbf{Z}$, or an element of $S = \mathbf{Z}/n\mathbf{Z}$ for a multiple or divisor n of m (with $R = \mathbf{Z}/m\mathbf{Z}$).

One(R)	Identity(R)

Zero(R)	Representative(R)

> These generic functions (cf. Chapter 17) create 1, 1, 0, and 0 respectively, in any $\mathbf{Z}/m\mathbf{Z}$.

Random(R)

> Create a "random" residue class in R.

19.5.2   Arithmetic Operators

+ n		- n

m + n	m - n	m * n	n ^ k	m / n	m div n

m +:= n	m -:= n	m *:= n	m /:= n	m ^:= k

19.5.3   Equality and Membership

m eq n	m ne n
n in R	n notin R

19.5.4   Parent and Category

Parent(n)	Category(n)

19.5.5   Predicates on Ring Elements

IsZero(n)	IsOne(n)	IsMinusOne(n)

IsNilpotent(n)	IsIdempotent(n)

IsUnit(n)	IsZeroDivisor(n)	IsRegular(n)

IsIrreducible(n)	IsPrime(n)

19.5.6   Solving Equations over $\mathbf{Z}/m\mathbf{Z}$

Solution(a, b)

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

IsSquare(n)

Factorization   [<RngIntElt, RngIntElt>]

> Given an element $n \in \mathbf{Z}/m\mathbf{Z}$ this function returns true if there exists $a \in \mathbf{Z}/m\mathbf{Z}$ such that $a^2 = n \in \mathbf{Z}/m\mathbf{Z}$, 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)

SquareRoot(a)

> Factorization [<RNGINTELT, RNGINTELT>]

Given an element a of the ring $\mathbf{Z}/m\mathbf{Z}$, this function returns an element b of $\mathbf{Z}/m\mathbf{Z}$ such that $b^2 = a \in \mathbf{Z}/m\mathbf{Z}$, 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)

AllSqrts(a)

> Factorization [<RNGINTELT, RNGINTELT>]

Return a sequence containing all square roots of the element a in a residue class ring $\mathbf{Z}/m\mathbf{Z}$. 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 H19E4

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!

19.6   Ideal Operations

$\boxed{\texttt{ideal< R | a}_1\texttt{, ..., a}_r \texttt{ >}}$

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

$\boxed{\texttt{GreatestCommonDivisor(a, b)}}$
$\boxed{\texttt{Gcd(a, b)}}$
$\boxed{\texttt{GCD(a, b)}}$

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

$\boxed{\texttt{GreatestCommonDivisor(Q)}}$
$\boxed{\texttt{Gcd(Q)}}$
$\boxed{\texttt{GCD(Q)}}$

Greatest common divisor of the sequence of elements Q, that is, a generator for the R-ideal generated by the elements in Q.

$\boxed{\texttt{LeastCommonMultiple(a, b)}}$
$\boxed{\texttt{Lcm(a, b)}}$
$\boxed{\texttt{LCM(a, b)}}$

Least common multiple of the elements a and b of R, that is, a generator for the R-ideal $(a) \cap (b)$.

$\boxed{\texttt{LeastCommonMultiple(Q)}}$
$\boxed{\texttt{Lcm(Q)}}$
$\boxed{\texttt{LCM(Q)}}$

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.

$\boxed{\texttt{I + J}}$ $\boxed{\texttt{I * J}}$ $\boxed{\texttt{I meet J}}$

$\boxed{\texttt{a in I}}$ $\boxed{\texttt{a notin I}}$

$\boxed{\texttt{I eq J}}$ $\boxed{\texttt{I ne J}}$

$\boxed{\texttt{I subset J}}$ $\boxed{\texttt{I notsubset J}}$

19.7   The Unit Group

`UnitGroup(R)`

Given $R = \mathbf{Z}/m\mathbf{Z}$, 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)`

Returns `true` if the element $n \in \mathbf{Z}/m\mathbf{Z}$ is primitive, that is, if it generates the multiplicative group of $\mathbf{Z}/m\mathbf{Z}$, `false` otherwise.

`PrimitiveElement(R)`

`PrimitiveRoot(R)`

Given $R = \mathbf{Z}/m\mathbf{Z}$, 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)`

Given an element a belonging to $\mathbf{Z}/m\mathbf{Z}$, return the multiplicative order $k \geq 1$ of a if a is in the unit group $(\mathbf{Z}/m\mathbf{Z})^*$, and zero if a is not a unit.

`Normalize(x)`

`Normalise(x)`

Given an element $x \in R = \mathbf{Z}/m\mathbf{Z}$, 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 H19E5_____

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 H19E6_____

We construct a residue class ring $R = \mathbf{Z}/m\mathbf{Z}$ 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
```

_____

19.8   Dirichlet Characters

Let R be a ring. Then a *Dirichlet character over R of modulus N* is a homomorphism

$$\varepsilon : (\mathbf{Z}/N\mathbf{Z})^* \to R^*,$$

where R^* is the group of invertible elements of R. We extend ε to a set theoretic map on the whole of $\mathbf{Z}$ 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 $(\mathbf{Z}/N\mathbf{Z})^* \to R^*$ factors through $(\mathbf{Z}/M\mathbf{Z})^*$ via the natural map $(\mathbf{Z}/N\mathbf{Z})^* \to (\mathbf{Z}/M\mathbf{Z})^*$.

19.8.1 Creation

DirichletGroup(N)

> The group of Dirichlet characters modulo N with values in `RationalField()`. (Note that this is a group of exponent at most 2.)

FullDirichletGroup(N)

> 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)

> 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)

> 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)

BaseExtend(G, R, z)

> 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)

> Assign names to the generators of the Dirichlet group G.

19.8.2 Element Creation

Elements(G)

> A sequence containing all Dirichlet characters in the Dirichlet group G.

Random(G)

> A random element of the Dirichlet group G.

G . i

> The ith generator of the group G.

G ! x

> 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)
```
```
KroneckerCharacter(D, R)
```

>   The Kronecker character $n \mapsto (d/n)$, where d is the fundamental discriminant asso-
>   ciated to the integer D.
>   When a ring R is given, this is returned as a character with values in R.

19.8.3   Attributes of Dirichlet Groups

```
BaseRing(G)
```

>   The ring in which characters in G take values.

```
Modulus(G)
```

>   The integer N such that G is a group of Dirichlet characters on $\mathbf{Z}/N$.

```
Order(G)
```

>   The order of the Dirichlet group G.

```
Exponent(G)
```

>   The exponent of the Dirichlet group G.

```
NumberOfGenerators(G)
```

>   The number of generators of the Dirichlet group G.

```
Generators(G)
```

>   A sequence containing generators for the Dirichlet group G.

```
UnitGenerators(G)
```

>   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)
```
```
GaloisConjugacyRepresentatives(seq)
```

>   This returns a sequence containing one representative from each Galois conjugacy
>   class (over $\mathbf{Q}$) of characters corresponding to a character in the given group or the
>   given sequence.

```
AbelianGroup(G)
```

>   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.

19.8.4 Attributes of Elements

`BaseRing(chi)`

> The ring in which the Dirichlet character χ takes values.

`Modulus(chi)`

> The modulus of the group of Dirichlet characters that contains χ.

`Conductor(chi)`

> 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)`

> A sequence of integers specifying the Dirichlet character χ (in terms of generators of the group containing χ).

`x eq y`

> Return `true` iff the given characters have the same modulus and values.

`Order(chi)`

> The order of the given element χ in a group of Dirichlet characters.

`IsTrivial(chi)`

> Returns `true` if and only if the Dirichlet character χ has order 1.

`IsPrimitive(chi)`

> Returns `true` iff the Dirichlet character χ is primitive (equivalently, if its conductor equals its modulus).

`AssociatedPrimitiveCharacter(chi)`

> The primitive character modulo the conductor of χ which takes the same values (on units) as χ.

`IsEven(chi)`

> 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)`

> 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)

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)

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)

The returns a character which is the same as χ, except which takes values in the smallest possible subring of the base ring of χ.

19.8.5   Evaluation

Evaluate(chi,n)

chi(n)

The value of the Dirichlet character χ at the integer n.

ValueList(chi)

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)

A sequence containing the values of χ on the ordered sequence of elements of $\mathbf{Z}/m$ given by `UnitGenerators(Parent(chi))`, where m is the modulus of χ.

OrderOfRootOfUnity(r, n)

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.)

19.8.6 Arithmetic

---
x * y
---
x / y

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

The Dirichlet character x raised to the power of n, where n is any integer.

---
x ^ phi

The image of the Dirichlet character x under the automorphism ϕ.

Sqrt(x)

Given a Dirichlet character x of odd order, this returns a square root of x (in the same group).

19.8.7 Example

Example H19E7

We begin by constructing the group of characters $(\mathbf{Z}/5\mathbf{Z})^* \to \mathbf{Q}^*$.

```
> 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;
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 $\mathbf{Q}(\sqrt{D})$. 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)
```

20 RATIONAL FIELD

Chapter 20

RATIONAL FIELD

20.1 Introduction

This Chapter describes functions relating to the field of rational numbers **Q**. Note that most functions for rational integers can be found in Chapter 18.

The rational field **Q** is automatically created when Magma is started up. That means that in **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.

20.1.1 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).

20.1.2 Coercion

The tables in Chapter 17 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 **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 **Q** and R (usually R, unless R is a subring of **Q** such as **Z**). The most important exceptions to the above rule are those cases where the result would lie in a structure strictly larger than both **Q** and R. Examples of this are $R = \mathbf{Z}[x]$, and the result would generally be in $\mathbf{Q}(x)$, and $R = O_K$, an order in a number field (and the result could be in K).

Example H20E1_____

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
>  R<x> := PolynomialRing(Integers());
>  // The following produces an error:
>  1/2 + x;
>> 1/2 + x;
       ^
Runtime error in '+': Bad argument types
```

20.1.3   Homomorphisms

Since homomorphisms are generally only allowed to be unitary, the specification of ring homomorphisms from $Q = \mathbf{Q}$ 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 | >
```

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 H20E2_____

Suppose we wish to coerce rational numbers with denominator not divisible by 11 into the ring $\mathbf{Z}/11\mathbf{Z}$ in the obvious way by sending r/s to $rs^{-1} \bmod 11$. 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
```

20.2 Creation Functions

20.2.1 Creation of Structures

The rational field **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()`
> `RationalField()`

> Create the field **Q** of rational numbers.

> `MaximalOrder(Q)`
> `IntegerRing(Q)`
> `IntegerRing()`
> `Integers()`
> `RingOfIntegers(Q)`

> Create the field **Z** of rational integers.

> `FieldOfFractions(Z)`

> The function `FieldOfFractions` returns the field **Q** when R is either the ring **Z** of rational integers, or the field **Q** itself.

> `Completion(Q, P)`

> Precision   RNGINTELT   *Default* : ∞

> 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.

20.2.2 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 **Q** or the parent ring **Z** first, since these structures are loaded whenever MAGMA is started up.

> `a / b`

> Given integers a and $b \neq 0$, form the rational number a/b (in reduced form). Of course a and be are allowed to be given as expressions defining integers.

> `Q ! [a]`

> The inverse function to `Eltseq`, returns `Q!a`.

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

>   Given the rational field **Q**, and integers a, b (with $b \neq 0$), construct the rational
>   number a/b, in reduced form.

```
Q ! a
```

>   Given the rational field **Q**, and an integer a, create the rational number $a = a/1$ in
>   **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)
```   ```
Identity(Q)
```
```
Zero(Q)
```   ```
Representative(Q)
```

>   These generic functions (cf. Chapter 17) create 1, 1, 0, and 0 respectively, in the
>   rational field **Q**.

```
RootOfUnity(n, Q)
```

>   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)
```

>   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.

20.3   Structure Operations

20.3.1   Related Structures

```
Category(Q)
```
```
Parent(Q)
```   ```
PrimeField(Q)
```

```
IntegralBasis(Q)
```

>   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)
```

>   Return the least cyclotomic field containing the cyclotomic field element q; if q is
>   rational this returns the rational field.

MinimalField(S)

>   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)

>   In analogy to the number fields, returns the coefficient field of Q which will be **Q**.

Basis(Q)

AbsoluteBasis(Q)

>   A basis for Q as a **Q**-vector space, i.e. [1].

UnitGroup(Q)

>   The unit group of the maximal order of **Q** (i.e. of **Z**).

ClassGroup(Q)

>   The class group of the ring of integers **Z** of **Q** (which is trivial).

AutomorphismGroup(Q)

AutomorphismGroup(Q, Q)

>   The group of **Q** automorphisms of **Q**, ie. a trivial finitely presented group, the parent structure for **Q**-automorphisms and a map from the group to actual field automorphisms. In this case, of course the only **Q**-automorphism will be the identity.

Algebra(Q, Q)

>   The field of the rational number form canonically an algebra. This function returns an associative **Q**-algebra isomorphic to **Q** and the map from the algebra to **Q**.

VectorSpace(Q, Q)

>   The field of the rational number form canonically a vector space. This function returns a **Q**-vector space isomorphic to **Q** and the map from the vector space to **Q**.

Decomposition(Q, p)

>   For a prime p or for the "infinite prime" Infinity() compute the decomposition in **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.

20.3.2 Numerical Invariants

The functions below are defined for the rational field $\mathbf{Q}$ mainly because it often arises as a degenerate case of quadratic or cyclotomic field constructions. See the corresponding Chapters 36 and 37 for more.

```
Characteristic(Q)
```

```
Conductor(Q)
```

The smallest positive integer n such that Q is contained in the cyclotomic field $\mathbf{Q}(\zeta_n)$. For the rational field this is 1.

```
Degree(Q)
```
```
AbsoluteDegree(Q)
```

The degree of Q as a number field (which is 1 for the rational field).

```
Discriminant(Q)
```
```
AbsoluteDiscriminant(Q)
```

The field discriminant of Q (which is 1 for the rational field).

```
DefiningPolynomial(Q)
```

An irreducible polynomial over $\mathbf{Q}$ a root of which generates Q as a number field (for the rational field this returns the linear polynomial $x - 1$).

```
Signature(Q)
```

The signature (number of real embeddings and pairs of complex embeddings) of $\mathbf{Q}$.

20.3.3 Ring Predicates and Booleans

```
IsCommutative(Q)        IsUnitary(Q)
```
```
IsFinite(Q)        IsOrdered(Q)
```
```
IsField(Q)        IsEuclideanDomain(Q)
```
```
IsPID(Q)        IsUFD(Q)
```
```
IsDivisionRing(Q)        IsEuclideanRing(Q)
```
```
IsPrincipalIdealRing(Q)        IsDomain(Q)
```
```
Q eq R        Q ne R
```

20.4   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.

20.4.1   Parent and Category

Parent(r)		Category(r)

20.4.2   Arithmetic Operators

+ a		- a							
a + b		a - b		a * b		a ^ k		a / b	
a +:= b		a -:= b		a *:= b		a /:= b		a ^:= k	

20.4.3   Numerator and Denominator

Numerator(q)

> The (integer) numerator of the rational number q in reduced form.

Denominator(q)

> The (integer) denominator of the rational number q in reduced form. This will always be a positive integer.

Example H20E3

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
```

20.4.4   Equality and Membership

a eq b		a ne b
a in R		a notin R

20.4.5   Predicates on Ring Elements

IsIntegral(q)

> Returns `true` if the rational number q is an element of the ring of integers, `false` otherwise.

IsZero(a)   IsOne(a)   IsMinusOne(a)

IsNilpotent(a)   IsIdempotent(a)

IsUnit(a)   IsZeroDivisor(a)   IsRegular(a)

IsIrreducible(a)   IsPrime(a)

20.4.6   Comparison

a gt b   a ge b   a lt b   a le b

Maximum(a, b)   Maximum(Q)

Minimum(a, b)   Minimum(Q)

20.4.7   Conjugates, Norm and Trace

ComplexConjugate(q)

> The complex conjugate of q, which will be the rational number q itself.

Conjugate(q)

> The conjugate of q, which will be the rational number q itself.

Norm(q)

Norm(q)

> The norm (in $\mathbf{Q}$) of q, which will be the rational number q itself.

Trace(q)

> The trace (in $\mathbf{Q}$) of q, which will be the rational number q itself.

MinimalPolynomial(q)

> 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.)

20.4.8   Absolute Value and Sign

AbsoluteValue(q)

Abs(q)

>   The absolute value $|q|$ of a rational number q.

Sign(q)

>   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)

>   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.

20.4.9   Rounding and Truncating

Ceiling(q)

>   The ceiling of the rational number q, that is, the least integer greater than or equal to q.

Floor(q)

>   The floor of the rational number q, that is, the largest integer less than or equal to q.

Round(q)

>   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)

>   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)

>   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.

20.4.10 Rational Reconstruction

Under certain circumstances it is useful to have a partial inverse of the function $\psi_m :$ $\mathbf{Q} \to \mathbf{Z}/m\mathbf{Z}$ 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 $s \in \mathbf{Z}/m\mathbf{Z}$ 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)

> 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.

20.4.11 Valuation

Valuation(x, p)

Valuation(x, I)

> 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 $x = p^v u$.

20.4.12 Sequence Conversions

ElementToSequence(a)

Eltseq(a)

> The sequence $[a]$ for compatibility with the other field types.

21 FINITE FIELDS

<div align="center">

Chapter 21
FINITE FIELDS

</div>

21.1 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].

21.1.1 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.

21.1.2 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!

21.1.3 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.

21.2 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.

21.2.1 Creation of Structures

FiniteField(q)

GaloisField(q)

GF(q)

   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)`

`GaloisField(p, n)`

`GF(p, n)`

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 >`

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 >
>
> | 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 >
>
> 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)
>
> 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)
>
> 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)
>
> 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 >
>
> | 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 >		
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)

BaseField(F)

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)

The subfield of F of prime cardinality.

IsPrimeField(F)

Returns whether field F is a prime field.

F meet G

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)

Given finite fields K and L, both of characteristic p, return the smallest field which contains both of them.

Example H21E1_____

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
```

21.2.2   Creating Relations

Embed(E, F)

> 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)

> 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.

21.2.3   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)

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)

AssertAttribute(F, "PowerPrinting", l)

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)

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")

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($\sim$F, [f])

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 $\sim$F to F in the call to this function.

> `Name(F, 1)`

Given a finite field F, return the element which has the name attached to it, that is, return the element `F.1` of F.

21.2.4   Homomorphisms

> `hom<  F -> G | x  >`

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.

21.2.5   Creation of Elements

> `F . 1`

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 >`
>
> `F ! a`

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 >
```

> 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 $a_i \in E$ ($0 \le i \le n-1$). 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)        Identity(F)

Zero(F)        Representative(F)
```

> These generic functions (cf. Chapter 17) create 1, 1, 0, and 0 respectively, in any finite field.

```
Random(F)
```

> Create a 'random' element of finite field F.

21.2.6   Special Elements

```
F . 1
```
```
Generator(F)
```

> 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)
```

> 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)
```

> 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)
```

> (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)

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)

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.

21.2.7   Sequence Conversions

SequenceToElement(s, F)

Seqelt(s, F)

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)

Eltseq(a)

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)

Eltseq(a, E)

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.

21.3   Structure Operations

21.3.1   Related Structures
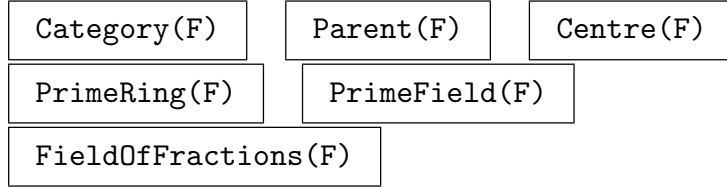
| Category(F) | Parent(F) | Centre(F) |

| PrimeRing(F) | PrimeField(F) |

| FieldOfFractions(F) |

| AdditiveGroup(F) |

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) |

| UnitGroup(F) |

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) |

Create the enumerated set consisting of the elements of finite field F.

| VectorSpace(F, E) |

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) |

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)

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)

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 H21E2_____

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)

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)

> 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.

21.3.2   Numerical Invariants

Characteristic(F)   # F

Degree(F)

> The absolute degree of F, that is, the degree over its prime subfield.

Degree(F, E)

> Given a finite field F that has been constructed as an extension of a field E, return the degree of F over E.

21.3.3   Defining Polynomial

DefiningPolynomial(F)

> 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)

> 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.

21.3.4   Ring Predicates and Booleans

IsConway(F)

> Given a finite field F, this function returns true iff F is defined over its prime field using a Conway polynomial.

IsDefault(F)

> Given a finite field F, this function returns true iff F is a default field.

IsCommutative(F)   IsUnitary(F)

IsFinite(F)   IsOrdered(F)

IsField(F)   IsEuclideanDomain(F)

IsPID(F)   IsUFD(F)

| IsDivisionRing(F) | IsEuclideanRing(F) |

| IsPrincipalIdealRing(F) | IsDomain(F) |

| F eq G | F ne G |

21.3.5   Roots

Roots(f)

>   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)

>   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)

>   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)

>   Return a primitive n-th root of unity in the smallest possible extension field of K.

Example H21E3_____

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 ]
```

21.4   Element Operations

See also Section 17.5.

21.4.1   Arithmetic Operators

+ a		− a

a + b		a − b		a * b		a / b

a ^ k

a +:= b		a −:= b		a *:= b

21.4.2   Equality and Membership

a eq b		a ne b

a in F		a notin F

21.4.3   Parent and Category

Parent(a)		Category(a)

21.4.4   Predicates on Ring Elements

IsZero(a)   IsOne(a)   IsMinusOne(a)

IsNilpotent(a)   IsIdempotent(a)

IsUnit(a)   IsZeroDivisor(a)   IsRegular(a)

IsIrreducible(a)   IsPrime(a)

IsPrimitive(a)

> 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)

> 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)

> 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)

> 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)

> 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.

21.4.5   Minimal and Characteristic Polynomial

MinimalPolynomial(a)

> 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)

> 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)

> 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)

> 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.)

21.4.6   Norm, Trace and Frobenius

Norm(a)

> The norm of the element a from the field F to the ground field of F.

Norm(a, E)

> 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)

NormAbs(a)

> The absolute norm of the element a, that is, the norm to the prime subfield of the parent field F of a.

Trace(a)

> The trace of the element a from the field F to the ground field of F.

Trace(a, E)

> 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)

TraceAbs(a)

> The trace of the element a, that is, the trace to the prime subfield of the parent field F of a.

Frobenius(a)

> 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)

> 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)

> The Frobenius image of x w.r.t. E; i.e., $x^{\#E}$.

Frobenius(a, E, r)

> The Frobenius image of x w.r.t. E; i.e., $x^{(\#E)^r}$.

NormEquation(K, y)

> 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)

> 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)

> 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.

21.4.7   Order and Roots

Order(a)

> The multiplicative order of the non-zero element a of the field F.

FactoredOrder(a)

> The multiplicative order of the non-zero element a of the field F as a factorization sequence.

SquareRoot(a)

Sqrt(a)

> 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)

> 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)

>   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)

>   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 H21E4

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
> 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
```

21.5 Polynomials for Finite Fields

IrreduciblePolynomial(F, n)

> 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)

> 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)

> Given an integer n in the range $1 \le n \le 100000$, 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)

> Given an integer n in the range $4 \le n \le 12800$, 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)

> 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)

> 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)

> Given a prime p and an exponent $n \ge 1$, 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)

> Given a prime p and an exponent $n > 1$, return `true` and the Conway polynomial if it is known for the field $\mathbf{F}_p$, `false` otherwise.

21.6   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 H27E3 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 p is less than 2^{30}, 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 very small characteristic. This means that logarithms of individual elements can be computed immediately if a relevant table is present for the specific field. 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) *Large Characteristic, Non-prime* :

   In all other cases, the Pohlig-Hellman algorithm is used.

Log(x)

> 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)

> 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)

> 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)

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` in the chapter on sparse matrices for more information.

SetVerbose("FFLog", v)

> (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 H21E5_____

We demonstrate the Log function.

```
> F<z> := FiniteField(7^4);
> PrimitiveElement(F);
z;
> Log(z);
1
> Log(z^2);
2
> Log(z + 1);
419
> 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);
4295700317032218908392
Time: 5.400
> z^4295700317032218908392;
z + 1
> time Log(z + 1);
4295700317032218908392
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
```

21.7 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)

> 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)

> 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)

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 H21E6_____

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
```

21.8   Bibliography

[**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*, pages 312–323. Springer-Verlag, 1993. Proc. 12th Annual International Cryptology Conference, Santa Barbara, Ca, USA, August 16–20, 1992.

[**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*, pages 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*, pages 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*, pages 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.

22 NEARFIELDS

Chapter 22
NEARFIELDS

22.1 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 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, Lö69] 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° 17]).

22.2 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 $\mathcal{Z}(N) = \bigcap \{\mathcal{K}(N)^x \mid x \in N, \ x \neq 0\}$.

22.2.1 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.

22.3   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`.

22.3.1   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)

> 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)

> 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)

> The list of Dickson triples (p, h, v) for the prime p, where *hb* and *vb* are bounds on h and v.

Example H22E1 _____

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 ],
    [ 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 [Lü71] 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 $s \equiv e$ (mod v).

NumberOfVariants(q, v)

The number of non-isomorphic nearfields with Dickson pair (q, v).

NumberOfVariants(N)

The number of variants of the Dickson nearfield N.

VariantRepresentatives(q, v)

> Representatives for the variant parameter of nearfields with Dickson pair (q, v).

Example H22E2_____

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*)

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 H22E3_____

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
```

22.3.2 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)

Creates the nth Zassenhaus nearfield.

Example H22E4_____

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
```

22.4 Operations on Elements

22.4.1 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		- a

a + b	a - b	a * b	a / b

a ^ k

a +:= b	a -:= b	a *:= b

Inverse(a)

    The inverse of a.

22.4.2 Equality and Membership

a eq b	a ne b
a in N	a notin N

22.4.3 Parent and Category

Parent(a)	Category(a)

N ! x

Element(N, x)

    Create a nearfield element from a finite field element.

ElementToSequence(x)

    Create a sequence from an element x of a nearfield.

22.4.4 Predicates on Nearfield Elements

IsZero(a)	IsUnit(a)	IsIdentity(a)

Example H22E5_____

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);
> 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 H22E6_____

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
```

22.5   Operations on Nearfields

```
N eq M
```

```
N ne M
```

```
#N
```

```
Cardinality(N)
```

> The cardinality of the nearfield N.

```
Random(N)
```

> A random element of the nearfield N.

```
Identity(N)
```

> The multiplicative identity of the nearfield N.

```
Zero(N)
```

> The additive identity of the nearfield N.

```
PrimeField(N)
```

> The prime field of the nearfield N.

```
Kernel(N)
```

> Return the kernel of the nearfield N as a finite field.

22.6 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)

UnitGroup(GrpPerm, N)

UnitGroup(GrpPC, N)

The unit group of the nearfield N.

Example H22E7_____

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)

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)

AffineGroup(GrpPerm, N)

AffineGroup(GrpPC, N)

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)

The extended unit group of a Dickson nearfield.

22.7 Automorphisms

IsIsomorphic(N1, N2)

Test whether the regular nearfields N_1 and N_2 are isomorphic. If they are, return an isomorphism.

AutomorphismGroup(N)

The automorphism group A of the regular nearfield N and a map giving the action of A on N.

22.8   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*)

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 H22E8_____

```
> 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
```

22.8.1   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*) |

   Check   BoolElt   *Default* : `false`

      The Hughes plane based on the nearfield N.

Example H22E9_____

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> ]
> 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
```

22.9   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.

[**Lŏ69**]   Heinz Lüneburg. *Lectures on projective planes.* University of Illinois, Chicago, 1969.

[**Lŏ71**]   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.

23 UNIVARIATE POLYNOMIAL RINGS

Chapter 23

UNIVARIATE POLYNOMIAL RINGS

23.1 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.

23.1.1 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.

23.2 Creation Functions

23.2.1 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)`

> `PolynomialRing(R)`

> 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 H23E1_____

We demonstrate the difference between global and non-global rings. We first create the global univariate polynomial ring over **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
```

23.2.2 Print Options

The `AssignNames` and `Name` functions can be used to associate a name with the indeterminate of a polynomial ring after creation.

AssignNames($\sim$P, s)

>   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 $\sim$P to P in the call to this function.

Name(P, i)

>   Given a polynomial ring P, return the i-th indeterminate of P (as an element of P).

23.2.3   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

>   Return the indeterminate for the polynomial ring P, as an element of P.

---

elt< P | a$_0$, ..., a$_d$ >

>   Given a polynomial ring $P = R[x]$ and elements $a_0, \ldots, a_d$ coercible into the co-
>   efficient ring R, return the polynomial $a_0 + a_1 x_n + \cdots + a_d x_n^d$ as an element of
> P.

P ! s

---

elt< P | s >

>   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)

> 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)

> 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)

> 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)   Identity(P)   Zero(P)   Representative(P)

Example H23E2_____

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
```

23.3   Structure Operations

23.3.1   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)

CoefficientRing(P)

CoefficientRing(f)

> Return the coefficient ring of polynomial ring P (the parent of f).

Category(P)		Parent(P)		PrimeRing(P)

23.3.2   Changing Rings

The `ChangeRing` function enables changing coefficient rings on a polynomial ring.

ChangeRing(P, S)

> 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)

> Given a polynomial ring $P = R[x]$, together with a ring S and a map $f : R \rightarrow S$, 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 H23E3 _____

In the first example of `ChangeRing` below we use automatic coercion of integers to rationals to go from $\mathbf{Z}[x]$ to $\mathbf{Q}[y]$. 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 $\mathbf{Z}$ in $\mathbf{Q}$.

```
> 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
```

23.3.3   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)

> 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

> 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)

23.3.4   Ring Predicates and Booleans

The usual ring functions returning Boolean values are available on polynomial rings.

IsCommutative(P)   IsUnitary(P)   IsFinite(P)   IsOrdered(P)

IsField(P)   IsEuclideanDomain(P)   IsPID(P)   IsUFD(P)

IsDivisionRing(P)   IsEuclideanRing(P)   IsDomain(P)

IsPrincipalIdealRing(P)   P eq Q   P ne Q   P lt Q

P gt Q   P le Q   P ge Q

23.3.5   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 >
---
hom< P -> S | y >

> 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 H23E4

In this example we map $\mathbf{Z}[x]$ 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
```

23.4   Element Operations

The categories for elements in univariate polynomial rings and their quotients are are `RngUPolElt` and `RngUPolResElt`

23.4.1   Parent and Category

Parent(p)	Category(p)

23.4.2   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	− a

a + b	a − b	a * b	a ^ k	a / b

a div b	a mod b

a +:= b	a −:= b	a *:= b

23.4.3   Equality and Membership

a eq b	a ne b

a in R	a notin R

23.4.4 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) | IsOne(a) | IsMinusOne(a) |

| IsNilpotent(a) | IsIdempotent(a) |

| IsUnit(a) | IsZeroDivisor(a) | IsRegular(a) |

| IsIrreducible(a) | IsPrime(a) | IsMonic(a) |

23.4.5 Coefficients and Terms

Coefficients(p)

ElementToSequence(p)

Eltseq(p)

> The coefficients of the polynomial $p \in R[x]$ in ascending order, as a sequence of elements of R.

Coefficient(p, i)

> 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)

> 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)

> 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)

> 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)

> Return the constant term, ie. the coefficient of x^0 as an element of the coefficient ring R.

> **Terms(p)**

> 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)**

> 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)**

> 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)**

> 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)**

> 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)**

> Given a polynomial $p \in P = R[x]$ where R is a subring of the real field (the ring of integers $\mathbf{Z}$, the rational field $\mathbf{Q}$, or a real field), return the polynomial in $Z[x]$ obtained from p by rounding all the coefficients of p.

> **Valuation(p)**

> 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 ∞.

23.4.6   Degree

> **Degree(p)**

> 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.

23.4.7 Roots

Roots(p)

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)

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)

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)

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)

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 H23E5_____

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)
```

(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).

23.4.8   Derivative, Integral

```
Derivative(p)
```

Given a polynomial $p \in P$, return the derivative of p as an element of P.

```
Derivative(p, n)
```

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)
```

Given a polynomial $p \in P$ over a field of characteristic zero, return the formal
integral of p as an element of P.

23.4.9   Evaluation, Interpolation

```
Evaluate(p, r)
```

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)
```

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 $1 \leq i \leq n$.

23.4.10   Quotient and Remainder

```
Quotrem(f, g)
```

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 ($R = \mathbf{Z}$) this
will be true in general when the leading coefficient of g divides that of f.

---

f div g

The quotient of the polynomial f by g, which is the first return value of `Quotrem` described above.

IsDivisibleBy(a, b)

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)

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

The remainder of division of the polynomial f by g, which is the second return value of `Quotrem` described above.

Valuation(f, g)

The exponent of the highest power of the polynomial g which divides the polynomial f.

Reductum(f)

The reductum of a polynomial f, which is the polynomial obtained by removing the leading term of f.

PseudoRemainder(f, g)

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)

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.

23.4.11 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)`

> 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)`

`CRT(X, M)`

> 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.

23.4.12 Other Operations

`ReciprocalPolynomial(f)`

> The reciprocal of the given univariate polynomial.

`PowerPolynomial(f,n)`

> The polynomial whose roots are the nth powers of the roots of the given polynomial (which should have coefficients in some field).

`f ^ M`

> 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).

23.5 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.

23.5.1   Common Divisors and Common Multiples

GreatestCommonDivisor(f, g)

Gcd(f, g)

GCD(f, g)

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)

Xgcd(f, g)

XGCD(f, g)

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)

Lcm(f, g)

LCM(f, g)

> 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)

> 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 $\mathbf{Z}$ 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.

23.5.2   Content and Primitive Part

Content(p)

> The content of p, that is, the greatest common divisor of the coefficients of p as an element of the coefficient ring.

PrimitivePart(p)

> The primitive part of p, being p divided by the content of p.

ContentAndPrimitivePart(p)

Contpp(p)

> 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.

23.6   Polynomials over the Integers

The functions in this section are available for univariate polynomials over the integers only.

Sign(p)

> The sign of the leading coefficient of p.

AbsoluteValue(p)

Abs(p)

> Returns either p or $-p$ according to which one has non-negative leading coefficient.

MaxNorm(p)

> The maximum of the absolute values of the coefficients of p.

SumNorm(p)

> The sum of the coefficients of p.

DedekindTest(p, m)

> 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].

23.7   Polynomials over Finite Fields

The functions in this section are available for univariate polynomials over finite fields only.

PrimePolynomials(R, d)

PrimePolynomials(R, d, n)

> 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)

> A random monic prime polynomial of R of degree d.

NumberOfPrimePolynomials(q, d)

NumberOfPrimePolynomials(K, d)

NumberOfPrimePolynomials(R, d)

> The number of monic prime polynomials of degree d over the respective finite field.

JacobiSymbol(a,b)

> The Jacobi symbol (a/b) of the two polynomials $a, b \in \mathbf{F}_q[x]$ 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.

23.8   Factorization

This section describes the functions for polynomial factorization and associated computations. These are available for several kinds of coefficient rings.

23.8.1   Factorization and Irreducibility

Factorization(f)

Factorisation(f)

    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 $\mathbf{F}_q$, the ring of integers $\mathbf{Z}$, the field of rationals $\mathbf{Q}$, an algebraic number field $\mathbf{Q}(\alpha)$, 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 ([vzGS92, KS95, Sho95]) 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 [vH02, vH01] 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)

Given a ring R, return whether factorization of polynomials over R is allowed in MAGMA.

SetVerbose("PolyFact", v)

(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)

Facpol(f)

Given a sequence of tuples, each consisting of pairs of irreducible polynomials and positive integer exponents, return the product polynomial.

Example H23E6_____

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 $\mathbf{Z}$.

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 $\mathbf{Z}[x]$, has degree 2^n, is irreducible over $\mathbf{Z}$, 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 `H42E2` 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;
>     "";
> 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)]);
1771080720430629161685158978892152599456 11
> 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)]);
85783447140360187781662744163364252674665633803596496806969 24587\
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)]);
461780752330314415975198835361983723394867968005788599782062 5979\
481789171112550210109817070112666284891955285248592492005163008
31
> time L := Factorization(p);
Time: 16.840
> #L;
2
> L[1,1] eq sd6;
true
> L[2,1] eq sd7;
true
```

See also Example H42E2 in the chapter on algebraically closed fields for a generalization of the
Swinnerton-Dyer polynomials.

SquarefreeFactorization(f)

Given a univariate polynomial f over the ring R, this function returns the square-
free 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)

   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)

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)

Given a univariate polynomial f over the ring R, this function returns 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)

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)

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.

23.8.2 Resultant and Discriminant

Discriminant(f)

The discriminant D of $f \in R[x]$ is returned. The discriminant is an element of R that can be defined by $D = c_n^{2n-2} \prod_{i \neq j}(\alpha_i - \alpha_j)$, where c_n is the leading coefficient of f and the α_i are the zeros of f (in some algebraic closure of R). The coefficient ring R must be a domain.

Resultant(f, g)

The resultant of univariate polynomials f and g (of degree m and n) in $R[x]$, which is by definition the determinant of the Sylvester matrix for f and g (a matrix of rank $m + n$ containing coefficients of f and g as entries). The resultant is an element of R. The coefficient ring R must be a domain.

> CompanionMatrix(f)

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

$$
\begin{pmatrix}
0 & 1 & 0 & \cdots & 0 \\
0 & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \ddots & & \vdots \\
-a_0 & -a_1 & -a_2 & \cdots - a_{d-2} & -a_{d-1}
\end{pmatrix}.
$$

23.8.3   Hensel Lifting

> HenselLift(f, s, P)

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 $\mathbf{Z}/p^k\mathbf{Z}$. The factorization of f modulo p must be squarefree, that is, s should not contain repeated factors.

Example H23E7_____

```
> 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
```

23.9   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.

23.9.1   Creation of Ideals and Quotients

```
ideal<  R | a₁, ..., aᵣ  >
```

> 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  >
```
```
quo<  R | a₁, ..., aᵣ  >
```

> 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 >;`.

23.9.2   Ideal Arithmetic

Since ideals of R are regarded as subrings of R, the ring R itself is a valid ideal as well.

```
I + J
```

> 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
```

> 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
```

> 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

> 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

> 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

> 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

> 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

> 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

> 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.

23.9.3   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

> Given an ideal I in a univariate polynomial ring R, return the generator of I in R as an element of I.

23.9.4   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)

> 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)

> If Q is the quotient $Q = R/I$ for some univariate polynomial ring R, this function returns R.

23.10   Special Families of Polynomials

23.10.1   Orthogonal Polynomials

ChebyshevFirst(n)

ChebyshevT(n)

> 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)

ChebyshevU(n)

> 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)

> 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)

LaguerrePolynomial(n, m)

> 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)

> 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)

> 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
>
> $$C_0^m(x) = 1, \; C_1^m(x) = 2mx,$$
> $$C_n^m(x) = \frac{1}{n}(2(n - 1 + m)xC_{n-1}^m(x) - (n + 2m - 2)C_{n-2}^m(x)).$$

23.10.2   Permutation Polynomials

DicksonFirst(n, a)

> 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)

> 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
>
> $$E_n(x, a) = \sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n - i}{i}(-a)^i x^{n-2i}.$$

23.10.3   The Bernoulli Polynomial

BernoulliPolynomial(n)

Given a positive integer n, this function constructs the n-th Bernoulli polynomial.

23.10.4   Swinnerton-Dyer Polynomials

SwinnertonDyerPolynomial(n)

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 $\mathbf{Z}[x]$, has degree 2^n, and is irreducible over $\mathbf{Z}$.

See Example H23E6 above which explains more about this class of polynomials, and see also Example H42E2 in the chapter on algebraically closed fields to see how these polynomials are constructed and also for a generalization.

23.11   Bibliography

[**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.

[**Knu97**] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, Massachusetts, 3rd edition, 1997.

[**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*, pages 398–406. ACM, 1995.

[**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.

[**Sho95**]   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*, pages 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*, pages 142–146. Springer, 2001.

[**vH02**]   Mark van Hoeij. Factoring Polynomials and the knapsack problem. *J. Number Th.*, 95(2):167–189, 2002.
URL: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.

24 MULTIVARIATE POLYNOMIAL RINGS

Chapter 24

MULTIVARIATE POLYNOMIAL RINGS

24.1 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 111 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 116. See also the chapters on affine algebras (Chapter 114) and on modules over affine algebras (Chapter 115).

24.1.1 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 111.2).

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 23 which devoted to univariate polynomial rings.)

24.2 Polynomial Rings and Polynomials

24.2.1 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 111.2 for details).

PolynomialRing(R, n)

PolynomialAlgebra(R, n)

   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 111.2). 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*)

PolynomialAlgebra(R, n, *order*)

Create a multivariate polynomial ring in $n > 0$ indeterminates over the ring R with the given order *order* on the monomials. See Section 111.2 for details.

Example H24E1 _____

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 H24E2_____

We demonstrate the difference between global and non-global rings. We first create the global multivariate polynomial ring over **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
```

24.2.2   Print Names

The `AssignNames` and `Name` functions can be used to associate names with the indeterminates of polynomial rings after creation.

AssignNames(∼P, s)

> 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)

> Given a polynomial ring P, return the i-th indeterminate of P (as an element of P).

24.2.3   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 111.3.2).

24.2.4   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 |

> Return the i-th indeterminate for the polynomial ring P in n variables ($1 \le i \le n$) as an element of P.

| elt< R | a > |

| R ! s |

| elt< R | s > |

> This element constructor can only be used for trivial purposes in multivariate polynomial rings: given a polynomial ring $P = R[x_1, \dots, x_n]$ 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) |

| MultivariatePolynomial(P, f, v) |

> Given a multivariate polynomial ring $P = R[x_1, \dots, x_n]$, 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 $1 \le i \le n$.
>
> The inverse operation is performed by the `UnivariatePolynomial` function.

| One(P) | | Identity(P) | | Zero(P) | | Representative(P) |

24.3   Structure Operations

24.3.1   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) |

| CoefficientRing(P) |

> Return the coefficient ring of polynomial ring P.

| Category(P) | | Parent(P) | | PrimeRing(P) |

24.3.2   Numerical Invariants

Note that the **#** operator only returns a value for finite (quotients of) polynomial rings.

> Rank(P)

> > Return the number of indeterminates of polynomial ring P over its coefficient ring.

> Characteristic(P)   #P

24.3.3   Ring Predicates and Booleans

The usual ring functions returning Boolean values are available on polynomial rings.

> IsCommutative(P)   IsUnitary(P)   IsFinite(P)   IsOrdered(P)

> IsField(P)   IsEuclideanDomain(P)   IsPID(P)   IsUFD(P)

> IsDivisionRing(P)   IsEuclideanRing(P)   IsDomain(P)

> IsPrincipalIdealRing(P)   P eq Q   P ne Q

24.3.4   Changing Coefficient Ring

The `ChangeRing` function enables the changing of the coefficient ring of a polynomial ring.

> ChangeRing(P, S)

> > 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.

24.3.5   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, y$_1$, ..., y$_n$ >
> hom< P -> S | y$_1$, ..., y$_n$ >

> > 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 H24E3_____

In this example we map $\mathbf{Q}[x, y]$ into the number field $\mathbf{Q}(\sqrt[3]{2}, \sqrt{5})$ by sending x to $\sqrt[3]{2}$ and y to $\sqrt{5}$ and the identity map on the coefficients (which we omit).

```
> Q := RationalField();
> 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
```

24.4   Element Operations

24.4.1   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 | - a |

| a + b | a - b | a * b | a ^ k | a / b | a div b |

| a +:= b | a -:= b | a *:= b | a div:= b |

24.4.2   Equality and Membership

| a eq b | a ne b |

| a in R | a notin R |

24.4.3 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)`

> 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)`

> Returns `true` iff the set S of multivariate polynomials is algebraically dependent.

`IsZero(f)`　　`IsOne(f)`　　`IsMinusOne(f)`

`IsNilpotent(f)`　　`IsIdempotent(f)`

`IsUnit(f)`　　`IsZeroDivisor(f)`　　`IsRegular(f)`

`IsIrreducible(f)`　　`IsPrime(f)`

24.4.4 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)`

> 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)`

`Coefficients(f, v)`

> 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 $1 \le i \le n$ 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)

Coefficient(f, v, k)

> 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 $1 \le i \le n$ 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)

> 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)

LeadingCoefficient(f, v)

> 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 $1 \le i \le n$ 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)

> Given a multivariate polynomial f, return the length of f, i.e., the number of terms of f.

TrailingCoefficient(f)

> 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)`

`TrailingCoefficient(f, v)`

> 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)`

`CoefficientNumerator(f)`

> Given a multivariate polynomial f with coefficients in $\mathbf{Q}$, 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)`

> 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)`

> 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)`

> 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)`

> 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)

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)

Terms(f, v)

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 $1 \le i \le n$ 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)

Term(f, v, k)

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the k-th term of f (with $k \ge 0$), 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 $1 \le i \le n$ 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)

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)

LeadingTerm(f, v)

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 $1 \le i \le n$ 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)

> 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)

TrailingTerm(f, v)

> 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)

> 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)

> 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)

> 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 H24E4 _____

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
]
> 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
```

24.4.5   Degrees

Degree(f, i)

Degree(f, v)

> 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)

> 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)

> 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.

24.4.6   Univariate Polynomials

IsUnivariate(f)

> 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)

IsUnivariate(f, v)

> 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)

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 H24E5_____

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
```

24.4.7 Derivative, Integral

```
Derivative(f, i)
```
```
Derivative(f, v)
```

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)
```
```
Derivative(f, k, v)
```

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)
```
```
Integral(f, v)
```

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 ] )
```

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.

24.4.8 Evaluation, Interpolation

```
Evaluate(f, s)
```

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)
```
```
Evaluate(f, v, r)
```

> 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)
```
```
Interpolation(I, V, v)
```

> 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 $1 \le i \le n$ that is the number of the variable or the variable v itself (as a polynomial).

Example H24E6_____

We define $P = \mathbf{Q}[x, y, z]$, 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
]
```

24.4.9   Quotient and Reductum

> [!NOTE] f div g
> [!NOTE] ExactQuotient(f, g)

The quotient of the multivariate polynomial f by g in $R[x_1, \ldots, x_n]$, provided the result lies in P again. Here R must be a domain. If a polynomial q in P exists such that $f = q \cdot g$ then it will be returned, but if does not exist an error results.

> [!NOTE] Reductum(f)

The reductum of a polynomial f, which is the polynomial obtained by removing the leading term of f.

> [!NOTE] Reductum(f, i)
> [!NOTE] Reductum(f, v)

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 $1 \le i \le n$.

24.4.10   Diagonalizing a Polynomial of Degree 2

We provide two basic tools that deal with polynomial diagonalization.

> [!NOTE] SymmetricBilinearForm(f)

The symmetric bilinear form (as a matrix) of a multivariate polynomial of degree 2.

> [!NOTE] DiagonalForm(f)

The diagonal form of the multivariate polynomial of degree 2. Also returns the transformation matrix.

Example H24E7

```
> 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 -
1765887328610189345243712105568836456192752173981161472348377104574041463371/2
```

```
>
> 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.

24.5   Greatest Common Divisors

The functions in this section can be applied to multivariate polynomials over any ring which has a GCD algorithm.

24.5.1   Common Divisors and Common Multiples

GreatestCommonDivisor(f, g)

Gcd(f, g)

GCD(f, g)

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 ([Wan80, MY73] 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 **Z** or most fields.

For polynomials over any finite field or any field of characteristic zero besides **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 uni-
variate GCD algorithms.

For polynomials over another polynomial ring or rational function field, the poly-
nomials 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)

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)
Lcm(f, g)
LCM(f, g)

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)

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)

Given a polynomial f over the base ring R, this function returns the unique nor-
malized 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 $\mathbf{Z}$ 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)
ClearDenominators(Q)

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.

24.5.2   Content and Primitive Part

Content(f)

> The content of f, that is, the greatest common divisor of the coefficients of f as an element of the coefficient ring.

PrimitivePart(f)

> The primitive part of f, being f divided by the content of f.

ContentAndPrimitivePart(f)

Contpp(f)

> 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.

24.6   Factorization and Irreducibility

We describe the functions for multivariate polynomial factorization and associated computations.

Factorization(f)

> 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 $\mathbf{F}_q$, the ring of integers $\mathbf{Z}$, the field of rationals $\mathbf{Q}$, an algebraic number field $\mathbf{Q}(\alpha)$, 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)

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)

Return the squarefree part of the multivariate polynomial f, which is the largest (normalized) divisor g of f which is squarefree.

IsIrreducible(f)

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)

(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 H24E8_____

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 H24E9_____

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;
>
> 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 H24E10

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 H24E11

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
```

24.7   Resultants and Discriminants

Resultant(f, g, i)

Resultant(f, g, v)

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. The coefficient ring R must be a domain. There are two ways to indicate with respect to which variable the integral is to be taken: either one specifies i, the integer $1 \le i \le n$ 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)

Discriminant(f, v)

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. There are two ways to indicate with respect to which variable the integral is to be taken: either one specifies i, the integer $1 \le i \le n$ 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).

24.8   Polynomials over the Integers

The functions in this section are available for multivariate polynomials over the integers
only.

Sign(f)

> The sign of the leading coefficient of f.

AbsoluteValue(f)

Abs(f)

> Return either f or $-f$ according to which one has non-negative leading coefficient.

MaxNorm(f)

> The maximum of the absolute values of the coefficients of f.

SumNorm(f)

> The sum of the base coefficients of f.

24.9   Bibliography

[**BM97**]   Laurent Bernardin and Michael B. Monagan.   Efficient Multivariate Factor-
ization Over Finite Fields.   In *Proceedings of AAECC*, volume 1255 of *LNCS*, pages
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*, pages 196–208. ACM press, 1976.

[**vH02**]   Mark van Hoeij. Factoring Polynomials and the knapsack problem. *J. Number
Th.*, 95(2):167–189, 2002.
URL: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.

25 REAL AND COMPLEX FIELDS

Chapter 25
REAL AND COMPLEX FIELDS

25.1 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 MFPR 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 MFPR 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.

25.1.1 Overview of Real Numbers in MAGMA

Real numbers are stored internally as expansions $\sum b_i 2^i$. 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 H25E1_____

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.

_____

25.1.2   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.

25.1.3   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 | >
```

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 H25E2_____

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
```

25.1.4   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)
```

> 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()
```

> Return the current parent for literal real numbers.

AssignNames($\sim$C, [s])

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 $\sim$C to C in the call to this function.

Name(C, 1)

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.

25.1.5   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.

25.2   Creation Functions

We describe the creation of real and complex fields and their elements.

25.2.1   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)

Bits	BOOLELT	*Default* : `false`

Given a positive integer p, create and return a version R of the real field $\mathbf{R}$ 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()

Return the default real field.

ComplexField(p)

Bits		BOOLELT		*Default* : `false`

Given a positive integer p, create and return a version C of the complex field $\mathbf{C}$ 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()

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)

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 H25E3_____

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
```

25.2.2   Creation of Elements

$\boxed{d \ . \ e\mathbf{e}f\mathbf{p}g}$

$\boxed{d \ . \ e\mathbf{E}f\mathbf{P}g}$

Given a succession of literal decimal digits d, a succession of literal decimal digits e, a succession of literal decimal digits f, and an integer g, construct the real number $r = d.e \times 10^f$. If specified, the effect of g is to create r as an element of the real field of precision g.

If g is omitted (together with $\mathbf{p}$ or $\mathbf{P}$), the real number will be created as an element of the default real field.

Both d and f may include a leading sign $\mathbf{+}$ or $\mathbf{-}$; leading zeroes in d and f are ignored. If e consists entirely of zeroes it may be omitted together with the $\mathbf{.}$ and if f is zero it may be omitted together with $\mathbf{E}$ (or $\mathbf{e}$). But note that if all of e, f and g are omitted the result will be an integer.

$\boxed{\texttt{elt< R | m, n >}}$

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.

$\boxed{\texttt{elt< C | x, y >}}$

$\boxed{\texttt{C ! [x, y]}}$

Given the complex field C and elements x and y coercible into the real field underlying C, construct the complex number $x + y\mathrm{i}$.

$\boxed{\texttt{R ! a}}$

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;

$\boxed{\texttt{C ! a}}$

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 H25E4

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);
1.2345000000000000000000000000 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.

One(R)	Identity(R)
Zero(R)	Representative(R)

25.3   Structure Operations

25.3.1   Related Structures

Category(R)	Parent(R)
PrimeField(R)	

25.3.2   Numerical Invariants

Characteristic(R)

25.3.3 Ring Predicates and Booleans

IsCommutative(R)	IsUnitary(R)

IsFinite(R)	IsOrdered(R)

IsField(R)	IsEuclideanDomain(R)

IsPID(R)	IsUFD(R)

IsDivisionRing(R)	IsEuclideanRing(R)

IsPrincipalIdealRing(R)	IsDomain(R)

R eq S	R ne S

25.3.4 Other Structure Functions

Precision(R)

> Return the decimal precision p to which calculations are performed in the real or complex field R.

BitPrecision(R)

> Return the (internally used) bit precision p to which calculations are performed in the real or complex field R.

25.4 Element Operations

25.4.1 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.

Parent(r)	Category(r)

IsZero(r)	IsOne(r)	IsMinusOne(r)

IsUnit(r)	IsZeroDivisor(r)

IsIdempotent(r)	IsNilpotent(r)

IsIrreducible(r)	IsPrime(r)

25.4.2   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.

The comparison functions `gt, ge, lt, le` are not defined for complex numbers.

a eq b	a ne b

a in R	a notin R

a gt b	a ge b	a lt b	a le b

Maximum(a, b)	Minimum(a, b)

Maximum(Q)	Minimum(Q)

25.4.3   Other Predicates

IsIntegral(c)

> Returns `true` if and only if the real or complex number c is a rational integer.

IsReal(c)

> 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.

25.4.4   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	- r

r + s	r - s	r * s	r / s	r ^ k

r +:= s	r -:= s	r *:= s	r /:= s	r ^:= s

25.4.5   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)

> 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)

> 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)

> 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)

Arg(c)

> 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.

Modulus(c)

> Given a complex number c, return the real number (to the same precision as c) that is the modulus of c.

Real(c)

Re(c)

> Given a complex number $c = x + y$i, return the real part x of c (as a real number to the same precision as c).

Imaginary(c)

Im(c)

> Given a complex number $c = x + y$i, return the imaginary part y of c (as a real number to the same precision as c).

25.4.6 Rounding

Round(r)

> 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)

> 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)
> ```
> ```
> Ceiling(r)
> ```

> The ceiling of the real number r, i.e. the smallest integer greater than or equal to r.

> ```
> Floor(r)
> ```
> ```
> Floor(r)
> ```

> The floor of the real number r, i.e. the greatest integer less than or equal to r.

25.4.7   Precision

> ```
> Precision(c)
> ```

> 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(c)
> ```

> 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)
> ```
> ```
> Precision(L)
> ```

> Gives a sequence of real or complex numbers, return the precision p of their parent field.

> ```
> ChangePrecision(r, n)
> ```
> ```
> ChangePrecision(c, n)
> ```

> Coerces the real (r) or complex (c) number into a field of precision n.

25.4.8   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)
> ```

> 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"*, for more information see `mpfr.org`.

---

*   `http://www-2.cs.cmu.edu/~adamchik/articles/catalan/catalan.htm`

EulerGamma(R)

> 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)

> The value of π computed to the precision of R.

25.4.9   Simple Element Functions

AbsoluteValue(r)
Abs(r)

> The absolute value of the real or complex number r.

Sign(r)

> Return one of the integer values $+1$, 0, -1 depending upon whether the real number r is positive, zero or negative, respectively.

ComplexConjugate(c)
Conjugate(c)

> The complex conjugate $x - y\mathrm{i}$ of a complex number $x + y\mathrm{i}$.

Norm(c)

> 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)

> 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)
Sqrt(c)

> 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)

> Max   ExtReElt   *Default : ∞*

> 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)

> Max   ExtReElt   *Default : ∞*

> 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.

25.4.10   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 $p = a_0 + a_1 z + \cdots + a_n z^n \in \mathbf{C}[z]$, 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 $\mathbf{C}[z]$ 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

$G = (z - u_{k+1}) \dots (z - u_n)$. Through shifts and scalings, we may assume that $\Gamma = \{c \in \mathbf{C} : \|z\| = 1\}$.

For m in $\{1, \dots, k\}$, let s_m denote the m-th power sum of the roots of p which lie inside the splitting circle. That is,

$$s_m = u_1^m + \dots + u_k^m.$$

The residue theorem can then be used to calculate s_m $(1 \le m \le k)$:

$$s_m = \frac{1}{2\pi i} \int_\Gamma z^m \frac{p'(z)}{p(z)} \, dz.$$

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 $p = F_0 G_0 + r$, 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

$$p - F_1 G_1 = p - F_0 G_0 - f G_0 - g F_0 - fg.$$

Hence choosing f and g such that

$$p - F_0 G_0 = f G_0 + g F_0$$

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

$$1 = H G_0 + L F_0,$$

where $\deg(H) < \deg(F_0)$ and $\deg(L) < \deg(G_0)$.

The polynomial H can be calculated using the formula

$$H(z) = \frac{1}{2\pi i} \int_\Gamma \frac{1}{(F_0 G_0)(t)} \frac{F_0(z) - F_0(t)}{z - t} \, dt.$$

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:

$$H_{m+1} \equiv H_m(2 - H_m G_0) \pmod{F_0}.$$

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

$$f \equiv H(p - F_0 G_0) \pmod{F_0}.$$

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

$$r_1(p) \leq r_2(p) \leq \cdots \leq r_n(p)$$

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

$$\frac{r_{k+1}(q)}{r_k(q)} > \exp\left(\frac{\Delta}{n-1}\right).$$

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)		
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 H25E5_____

```
> P<z> := PolynomialRing(ComplexField());
> p := (z-1.1)^6;
> p;
z^6 - 6.6000000000000000000000000000001*z^5 +
    18.150000000000000000000000000000*z^4 -
    26.620000000000000000000000000000*z^3 +
    21.961500000000000000000000000000*z^2 -
    9.6630600000000000000000000000003*z +
    1.7715610000000000000000000000001
> R := Roots(p);
> R;
[ <1.1000133059659060542165199857, 1>,
<1.1000066528943086096929868917 +
1.1523304495817982565148665125E-5*i, 1>,
<1.1000066528943086096929868917 -
1.1523304495817982565148665125E-5*i, 1>,
<1.0999866942113803052183473123 4, 1>,
<1.0999933470170482105895796553 7 +
1.1523150961326985800466916771 1E-5*i, 1>,
<1.0999933470170482105895796553 7 -
1.1523150961326985800466916771 1E-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)

> 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 H25E6

```
> P<z> := PolynomialRing(ComplexField());
> p := (z-1.1)^6;
> R, E := RootsNonExact(p);
> R;
[ 1.10001483296913451410370191006 -
8.56404454142796527111307103383E-6*i,
1.10001483296913451410370191006 +
8.56404454142796527111304767692E-6*i,
1.09998516742199321904393975959 +
8.56336708832137724325720239457E-6*i,
1.09999999960887226685235833026 +
1.71274116266516824685125851069E-5*i,
1.09998516742199321904393771623 -
8.56336708832137723945115457519E-6*i,
1.09999999960887226685236037365 -
1.71274116266516824723187272572E-5*i ]
> E;
[ 0.0048231441542156971991062164306,
0.0048231441542156971991062164306,
0.0048230140891973860561847686767,
0.0048230791126115946099162101745,
0.0048230140891973860561847686767,
0.0048230791126115946099162101745 ]
```

HenselLift(f, R, k)

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.

25.4.11 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)

  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

$$s_1 + \cfrac{1}{s_2 + \cfrac{1}{s_3 + \cdots + \cfrac{1}{s_n}}}.$$

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)

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)

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.

25.4.12   Linear and Algebraic Dependencies

LinearRelation(q: *parameters*)

LinearRelation(v: *parameters*)

>   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)

>   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)

IntegerRelation(q,N)

>   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 30.7). 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*)

>   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)

MinimalPolynomial(r,d)

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 30.7). 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 H25E7_____

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 23.8) relies upon Van Hoeij's algorithm [vH02, vH01]. 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];
```

```
> 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
```

25.5   Transcendental Functions

25.5.1   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)

> Given a power series f defined over a real or complex field, return the exponential power series of f.

Exp(c)

> 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)

> 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(c)

> 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)

Given non-negative real numbers b and r, return the logarithm $\log_b(r)$ of a to the base b. Automatic coercion is applied if necessary.

Dilog(s)

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)

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)

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)
> ```
> ```
> PolylogDold(m, s)
> ```
> ```
> PolylogP(m, s)
> ```
>
> 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 [Zag91]. PARI is used here.

25.5.2 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)
> ```
>
> Given a power series f defined over a real or complex field, return the power series $\sin(f)$.

> ```
> Sin(c)
> ```
>
> Given a real or complex number c, return the value $\sin(c)$.

> ```
> Cos(f)
> ```
>
> Given a power series f defined over a real or complex field, return the power series $\cos(f)$.

> ```
> Cos(c)
> ```
>
> Given a real or complex number c, return the value $\cos(c)$.

> ```
> Sincos(f)
> ```
>
> Given a power series f defined over a real or complex field, return the two power series $\sin(f)$ and $\cos(f)$.

> ```
> Sincos(s)
> ```
>
> Given a real or complex number s, return the two values $\sin(s)$ and $\cos(s)$.

Tan(f)

> Given a power series f defined over the real or complex field, return the power series $\tan(f)$.

Tan(c)

> 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)

> Given a power series f defined over a real or complex field having valuation zero, return the power series $\cot(f)$.

Cot(c)

> 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)

> Given a power series f defined over a real or complex field, return the power series $\sec(f)$.

Sec(c)

> 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)

> Given a power series f defined over a real or complex field having valuation zero, return the power series $\operatorname{cosec}(f)$.

Cosec(c)

> Given a real or complex number c, return the value $\operatorname{cosec}(c) = 1/\sin(c)$. Note that c should not be too close to one of the zeroes $n \cdot \pi$ of $\sin(z)$.

25.5.3   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

$$\arcsin(z) = \frac{1}{i} \log(iz + \sqrt{1 - z^2}),$$

$$\arccos(z) = \frac{1}{i} \log(z + \sqrt{z^2 - 1}),$$

$$\arctan(z) = \frac{1}{2i} \log\left(\frac{1 + iz}{1 - iz}\right).$$

> **Arcsin(f)**
>
> Given a power series f defined over a real or complex field. return the inverse sine of the power series f.

> **Arcsin(r)**
>
> 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)**
>
> Given a power series f defined over a real or complex field. return the inverse cosine of the power series f.

> **Arccos(r)**
>
> 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)**
>
> Given a power series f defined over the real or complex field, return the inverse tangent of the power series f.

Arctan(r)

> 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)

Arctan2(x, y)

> 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 x and $\sin(v)$ coincide, as well as the signs of y 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).

Arccot(r)

> 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)

> 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)

> 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.

25.5.4   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)

> Given a power series f defined over a real or complex field, return the hyperbolic sine of the power series f.

Sinh(s)

> 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)

> Given a power series f defined over a real or complex field, return the hyperbolic cosine of the power series f.

Cosh(r)

> 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)

> Given a power series f defined over a real or complex field, return the hyperbolic tangent of the power series f.

Tanh(r)

> 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)

> 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)

> Given a real or complex number s, return $\operatorname{sech}(s) = 1/\cosh(s)$. The result will be a real or complex value, in accordance with the argument.

Cosech(r)

> Given a real or complex number s, return $\operatorname{cosech}(s) = 1/\sinh(s)$. The result will be a real or complex value, in accordance with the argument.

25.5.5 Inverse Hyperbolic Functions

The inverse hyperbolic functions are available for real or complex arguments. The principal values are chosen as indicated.

Argsinh(f)

> Given a power series f defined over a real or complex field, return the inverse hyperbolic sine of the power series f.

Argsinh(r)

> 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)

Given a power series f defined over a real or complex field, return the inverse hyperbolic cosine of the power series f.

Argcosh(r)

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)

Given a power series f defined over a real or complex field, return the inverse hyperbolic tangent of the power series f.

Argtanh(s)

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)

Given a real or complex number s, return t such that $\mathrm{sech}(t)) = s$; 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)

Given a real or complex number s, return t such that $\mathrm{cosech}(t)) = s$; 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)

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.

25.6   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].

25.6.1   Eisenstein Series

Let $f(z)$ be a modular function. Then $f(z)$ may be written as a Fourier series

$$f(z) = \sum_{n \in \mathbf{Z}} a_n q^n,$$

where $q = e^{2\pi i z}$, 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 $\mathbf{C}$. 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)

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)

   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)

   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)

   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 $e^{2\pi * i * \tau}$.

Example H25E8_____

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;
1.00000000000000000000000000000 +
    240.000000000000000000000000000*q +
    2160.00000000000000000000000000*q^2 +
    6720.00000000000000000000000000*q^3 +
    17520.0000000000000000000000000*q^4 +
    30240.0000000000000000000000000*q^5 +
    60480.0000000000000000000000000*q^6 +
    82560.0000000000000000000000000*q^7 +
    140400.000000000000000000000000*q^8 +
    181680.000000000000000000000000*q^9 +
    272160.000000000000000000000000*q^10 +
    319680.000000000000000000000000*q^11 +
    490560.000000000000000000000000*q^12 +
    527520.000000000000000000000000*q^13 +
    743040.000000000000000000000000*q^14 +
    846720.000000000000000000000000*q^15 +
    1123440.00000000000000000000000*q^16 +
    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);
0.559302852856190773766762411942 +
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);
0.559302852856190773766762411942 +
3.67329046709782088758389413820E-31*i
```

25.6.2   Weierstrass Series

`WeierstrassSeries(z, q)`

    Precision   RNGINTELT   *Default :*

      Return a normalized q-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 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)`

      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)`

      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)`

      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 $\tau = \left(-b + \sqrt{b^2 - 4ac}\right)/(2a)$.

25.6.3   The Jacobi θ and Dedekind η-functions

The first Jacobi θ-function, $\theta(q, z)$, is defined by

$$\theta(q, z) = \frac{1}{i} \sum_{n=-\infty}^{\infty} (-1)^n q^{(n+\frac{1}{2})^2} e^{(2n+1)iz} = 2\sum_{n=0}^{\infty}(-1)^n q^{(n+\frac{1}{2})^2} \sin(2n + 1)z.$$

      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 $m_1\pi + m_2\frac{\log x}{i}$ for any integers m_1, m_2.

`JacobiTheta(q, z)`

      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)

> 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)

> 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)

> 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)

> 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 $q = e^{2\pi i s}$.

25.6.4 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)

> 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).

<div style="border:1px solid black; display:inline-block; padding:4px">jInvariant(s)</div>

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

$$j(s) = e^{-2\pi i s} + 744 + 196884 e^{2\pi i s} + \cdots.$$

<div style="border:1px solid black; display:inline-block; padding:4px">jInvariant(L)</div>

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.

<div style="border:1px solid black; display:inline-block; padding:4px">jInvariant(F)</div>

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 $\tau = \left(-b + \sqrt{b^2 - 4ac}\right)/(2a)$.

<div style="border:1px solid black; display:inline-block; padding:4px">Delta(z)</div>

Given a complex power series z, this function returns a q-series expansion of the discriminant $\Delta(z)$.

<div style="border:1px solid black; display:inline-block; padding:4px">Delta(t)</div>

Given a point t in the upper half plane, return the q-series expansion of the discriminant $\Delta(q)$ evaluated at $q = e^{2\pi i t}$.

<div style="border:1px solid black; display:inline-block; padding:4px">Delta(L)</div>

Given a pair $L = [a,b]$ of complex numbers generating a lattice in $\mathbf{C}$, 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.

25.6.5   Weber's Functions

<div style="border:1px solid black; display:inline-block; padding:4px">WeberF(s)</div>

For complex argument s in the upper half-plane, this returns the value of Weber's function f, defined in such a way that

$$j(s) = \frac{(f(s)^{24} - 16)^3}{f(s)^{24}}.$$

<div style="border:1px solid black; display:inline-block; padding:4px">WeberF2(g)</div>

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)

WeberF2(s)

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 H25E9_____

We compute the q-expansion for the Weber function $f_2(z)$.

```
> C<i> := ComplexField();
> R<x> := PowerSeriesRing(C);
> f2<q> := WeberF2(x);
> f2;
1.4142135623730950488016887242I +
    (1.4142135623730950488016887242I +
    0.37024024484653052058465674917I*i)*q +
    (1.3657492276533806075922612177I +
    0.37024024484653052058465674917I*i)*q^2 +
    (2.7799627900264756563939499419I +
    0.36601093379329241948227297708I*i)*q^3 +
    (2.7802395977876131340886473421I +
    0.7362511786398229400669297262I*i)*q^4 +
    (4.1459888254409937416809085598I +
    0.73626567226030370903683781952I*i)*q^5 +
    (5.5602017554105939807275554223I +
    1.1022766060535961285191107966I*i)*q^6 +
    ...
```

25.7   Theta Functions

One of the main tools for working with analytic Jacobians is the theta function. For instance it is used by `FromAnalyticJacobian` on page 10-4514 and `RosenhainInvariants` on page 10-4521. For $c \in \mathbf{R}^{2g}$ let c' be the first g entries and c'' the second g entries of c. For such a c, $z \in \mathbf{C}^g$ and τ an element of Siegel upper half-space the classical multi-variable theta function is defined by

$$\theta[c](z,\tau) = \sum_{m \in \mathbf{Z}^g} \exp(\pi i^t (m + c')\tau(m + c') + 2\pi i^t (m + c')(z + c'')).$$

The vector c is called the characteristic of the theta function.

| `Theta(char, z, tau)` |

> 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)` |

> 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.

25.8   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)` |

> 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)` |
| `Gamma(r)` |

> 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(r, s)

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)

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)

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)

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)

Psi(s)

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)

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

$$J_n(x) = \frac{1}{2\pi i}\left(\frac{z}{2}\right)^n\int_{-\infty}^{0^+} u^{-n-1}e^{u-\frac{z^2}{4t}}\,du,$$

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)

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)

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)

KBessel2(n, s)

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

$$K_n(s) = \frac{\pi}{2}\left(i^n J_{-n}(is) - i^{-n} J_n(s)\right)\cot(n\pi).$$

The function KBessel2 is an alternative (often faster) implementation of this function. PARI is used here.

25.9   The Hypergeometric Function

For more information on the Hypergeometric Series, see Husemöller [Hus87], page 176.

HypergeometricSeries(a,b,c, z)

Return the hypergeometric series $F(a, b, c; z)$ defined by

$$F(a, b, c; z) = \sum_{0 \leq n} \frac{(a)_n (b)_n}{n!(c)_n z^n}$$

where $(a)_n = a(a+1) \cdots (a+n-1)$.

HypergeometricU(a, b, s)

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.

25.10   Other Special Functions

ArithmeticGeometricMean(x, y)
AGM(f, g)

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)
AGM(x, y)

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)

> 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)

> 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)

> Given a real number r, compute the value of *Dawson's integral*,
> $$e^{-x^2} \cdot \int_0^x e^{u^2} du,$$
> at $x = r$. The mp real package is used here.

ErrorFunction(r)

Erf(r)

> 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)

Erfc(r)

> 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)

> 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)

> 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)

> Given a non-negative real number r that is not equal to 1, evaluate the *logarithmic integral* $y = \text{li}(x)$ at $x = r$. This integral is defined to be the principal value of
> $$\int_0^x \frac{1}{\log(u)} du.$$
> The mp real package is used here.

ZetaFunction(s)

ZetaFunction(R, n)

> 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 $\text{Re}(z) > 1$). 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].

25.11   Numerical Functions

This section contains some functions for numerical analysis, taken from PARI.

25.11.1   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 nth 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)

> 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)

> 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)

   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.

25.11.2   Integration

A number of 'Romberg-like' integration methods have been taken from Pari. The precision should not be made too large for this, and singularities are not allowed in the interval of integration (including its boundaries).

Interpolation(P, V, x)

> Using Neville's algorithm, interpolate the value of x under a polynomial p such that $p(P[i]) = V[i]$. An estimate of the error is also returned.

RombergQuadrature(f, a, b: *parameters*)

   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)

> Using Simpson's rule on n sub-intervals, approximate the integral of f from a to b.

TrapezoidalQuadrature(f, a, b, n)

> Using the trapezoidal rule on n sub-intervals, approximate the integral of f from a to b.

25.11.3 Numerical Derivatives

There is also a function to compute the NumericalDerivative of a function. This works via computing enough interpolation points and using a Taylor expansion.

NumericalDerivative(f, n, z)

> Given a suitably nice function f, compute a numerical approximation to the nth derivative at the point z.

Example H25E10 _____

```
> f := func<x|Exp(2*x)>;
> NumericalDerivative(f, 10, ComplexField(30)! 1.0) / f (1.0);
1024.00000000000000000000000000000
> NumericalDerivative(func<x|LogGamma(x)>,1,ComplexField()!3.0);
0.922784335098467139393487909918
> Psi(3.0); // Psi is Gamma'/Gamma
0.922784335098467139393487909918
```

25.12 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)*, pages 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.

[**Lew81**]   Leonard Lewin. *Polylogarithms and associated functions*. North Holland, New York, 1981.

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

[**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. URL: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.

[**vdGOS91**] G. van der Geer, F. Oort, and J. Steenbrink, editors. *Arithmetic Algebraic Geometry*, volume 89 of *Progress in Mathematics*, Basel, 1991. Birkhäuser Verlag.

[**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*, pages 142–146. Springer, 2001.

[**vH02**]   Mark van Hoeij. Factoring Polynomials and the knapsack problem. *J. Number Th.*, 95(2):167–189, 2002.
URL: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.

[**Zag91**]   Don Zagier. Polylogarithms, Dedekind Zeta Functions, and the Algebraic K-Theory of Fields. In van der Geer et al. [vdGOS91], pages 377–390.

PART IV
MATRICES AND LINEAR ALGEBRA

26 MATRICES

Chapter 26

MATRICES

26.1 Introduction

This chapter describes all the basic operations available for creating and working with matrices. Matrices arise in many different contexts and there are several types of matrix within MAGMA, but most of the operations listed here apply to all types of matrix.

The parent of any matrix will be one of several types of matrix-structure (module, matrix algebra, matrix group, etc.), and each of these matrix-structure types are described in other chapters, together with operations peculiar to their elements.

26.2 Creation of Matrices

This section describes the elementary constructs provided for creating a matrix or vector. For each of the following functions, the parent of the result will be as follows:

(a) If the result is a vector then its parent will be the appropriate R-space (of type `ModTupRng` or `ModTupFld`).

(b) If the result is a square matrix then its parent will be the appropriate matrix algebra (of type `AlgMatElt`).

(c) If the result is a non-square matrix then its parent will be the appropriate R-matrix space (of type `ModMatRng` or `ModMatFld`).

A matrix or a vector may also be created by coercing a sequence of ring elements into the appropriate parent matrix structure. There is also a virtual type `Mtrx` and all matrix types inherit from `Mtrx`. While writing package intrinsics, an argument should be declared to be of type `Mtrx` if it is a general matrix.

26.2.1 General Matrix Construction

`Matrix(R, m, n, Q)`

> Given a ring R, integers $m, n \geq 0$ and a sequence Q, return the $m \times n$ matrix over R whose entries are those specified by Q, coerced into R. Either of m and n may be 0, in which case Q must have length 0 (and may even be null), and the $m \times n$ zero matrix over R is returned. There are several possibilities for Q:

> (a) The sequence Q may be a sequence of length mn containing elements of a ring S, in which case the entries are given in row-major order. In this case, the function is equivalent to `MatrixRing(R, n)!Q` if $m = n$ and `RMatrixSpace(R, m, n)!Q` otherwise.

> (b) The sequence Q may be a sequence of tuples, each of the form `<i, j, x>`, where $1 \leq i \leq m$, $1 \leq j \leq n$, and $x \in S$ for some ring S. Such a tuple specifies that

the (i, j)-th entry of the matrix is x. If an entry position is not given then its value is zero, while if an entry position is repeated then the last value overrides any previous value(s). This case is useful for creating sparse matrices.

(c) The sequence Q may be a sequence of m sequences, each of length n and having entries in a ring S, in which case the rows of the matrix are specified by the inner sequences.

(d) The sequence Q may be a sequence of m vectors, each of length n and having entries in a ring S, in which case the rows of the matrix are specified by the vectors.

Example H26E1

This example demonstrates simple ways of creating matrices using the general `Matrix(R, m, n, Q)` function.

(a) Defining a 2×2 matrix over **Z**:

```
> X := Matrix(IntegerRing(), 2, 2, [1,2, 3,4]);
> X;
[1 2]
[3 4]
> Parent(X);
Full Matrix Algebra of degree 2 over Integer Ring
```

(b) Defining a 2×3 matrix over $\mathbf{F}_{23}$:

```
> X := Matrix(GF(23), 2, 3, [1,-2,3, 4,100,-6]);
> X;
[ 1 21  3]
[ 4  8 17]
> Parent(X);
Full KMatrixSpace of 2 by 3 matrices over GF(23)
```

(c) Defining a sparse 5×10 matrix over **Q**:

```
> X := Matrix(RationalField(), 5, 10, [<1,2,23>, <3,7,11>, <5,10,-1>]);
> X;
[ 0 23  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0 11  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0 -1]
> Parent(X);
Full KMatrixSpace of 5 by 10 matrices over Rational Field
```

(c) Defining a sparse 10×10 matrix over $\mathbf{F}_{101}$:

```
> X := Matrix(GF(101), 10, 10, [<2*i-1, 2*j-1, i*j>: i, j in [1..5]]);
> X;
[  1   0   2   0   3   0   4   0   5   0]
[  0   0   0   0   0   0   0   0   0   0]
```

```
[  2   0   4   0   6   0   8   0  10   0]
[  0   0   0   0   0   0   0   0   0   0]
[  3   0   6   0   9   0  12   0  15   0]
[  0   0   0   0   0   0   0   0   0   0]
[  4   0   8   0  12   0  16   0  20   0]
[  0   0   0   0   0   0   0   0   0   0]
[  5   0  10   0  15   0  20   0  25   0]
[  0   0   0   0   0   0   0   0   0   0]
> Parent(X);
Full Matrix Algebra of degree 10 over GF(101)
```

26.2.2   Shortcuts

The following functions are "shortcut" versions of the previous general creation function, where some of the arguments are omitted since they can be inferred by MAGMA.

Matrix(m, n, Q)

> Given integers $m, n \geq 0$ and a sequence Q of length mn containing elements of a ring R, return the $m \times n$ matrix over R whose entries are the entries of Q, in row-major order. Either of m and n may be 0, in which case Q must have length 0 and some universe R. This function is equivalent to MatrixRing(Universe(Q), n)!Q if $m = n$ and RMatrixSpace(Universe(Q), m, n)!Q otherwise.

Matrix(m, n, Q)

> Given integers m and n, and a sequence Q consisting of m sequences, each of length n and having entries in a ring R, return the $m \times n$ matrix over R whose rows are given by the inner sequences of Q.

Matrix(Q)

> Given a sequence Q of m vectors, each of length n over a ring R, return the $m \times n$ matrix over R whose rows are the entries of Q.

Matrix(R, n, Q)

> Given a ring R, an integer $n \geq 0$ and a sequence Q of length l containing elements of a ring S, such that n divides l, return the $(l/n) \times n$ matrix over R whose entries are the entries of Q, coerced into R, in row-major order. The argument n may be 0, in which case Q must have length 0 (and may even be null), in which case the 0×0 matrix over R is returned. This function is equivalent to MatrixRing(R, n)!Q if $l = n^2$ and RMatrixSpace(R, #Q div n, n)!Q otherwise.

Matrix(n, Q)

>   Given an integer $n \geq 0$ and a sequence Q of length l containing elements of a ring R, such that n divides l, return the $(l/n) \times n$ matrix over R whose entries are the entries of Q, in row-major order. The argument n may be 0, in which case Q must have length 0 and some universe R, in which case the 0×0 matrix over R is returned. This function is equivalent to `MatrixRing(Universe(Q), n)!Q` if $l = n^2$ and `RMatrixSpace(Universe(Q), #Q div n, n)!Q` otherwise.

Matrix(Q)

>   Given a sequence Q consisting of m sequences, each of length n and having entries in a ring R, return the $m \times n$ matrix over R whose rows are given by the inner sequences of Q.

Matrix(R, Q)

>   Given a sequence Q consisting of m sequences, each of length n and having entries in a ring S, return the $m \times n$ matrix over R whose rows are given by the inner sequences of Q, with the entries coerced into R.

Example H26E2_____

The first matrix in the previous example may be created thus:

```
> X := Matrix(2, [1,2, 3,4]);
> X;
[1 2]
[3 4]
> X := Matrix([[1,2], [3,4]]);
> X;
[1 2]
[3 4]
```

The second matrix in the previous example may be created thus:

```
> X := Matrix(GF(23), 3, [1,-2,3, 4,100,-6]);
> X;
[ 1 21  3]
[ 4  8 17]
> Parent(X);
Full KMatrixSpace of 2 by 3 matrices over GF(23)
> X := Matrix(GF(23), [[1,-2,3], [4,100,-6]]);
> X;
[ 1 21  3]
[ 4  8 17]
> X := Matrix([[GF(23)|1,-2,3], [4,100,-6]]);
> X;
[ 1 21  3]
[ 4  8 17]
```

26.2.3   Construction of Structured Matrices

ZeroMatrix(R, m, n)

Given a ring R and integers $m, n \geq 0$, return the $m \times n$ zero matrix over R.

ScalarMatrix(n, s)

Given an integer $n \geq 0$ and an element s of a ring R, return the $n \times n$ scalar matrix over R which has s on the diagonal and zeros elsewhere. The argument n may be 0, in which case the 0×0 matrix over R is returned. This function is equivalent to `MatrixRing(Parent(s), n)!s`.

ScalarMatrix(R, n, s)

Given a ring R, an integer $n \geq 0$ and an element s of a ring S, return the $n \times n$ scalar matrix over R which has s, coerced into R, on the diagonal and zeros elsewhere. n may be 0, in which case in which case the 0×0 matrix over R is returned. This function is equivalent to `MatrixRing(R, n)!s`.

DiagonalMatrix(R, n, Q)

Given a ring R, an integer $n \geq 0$ and a sequence Q of n ring elements, return the $n \times n$ diagonal matrix over R whose diagonal entries correspond to the entries of Q, coerced into R.

DiagonalMatrix(R, Q)

Given a ring R and a sequence Q of n ring elements, return the $n \times n$ diagonal matrix over R whose diagonal entries correspond to the entries of Q, coerced into R.

DiagonalMatrix(Q)

Given a sequence Q of n elements from a ring R, return the $n \times n$ diagonal matrix over R whose diagonal entries correspond to the entries of Q.

Matrix(A)

Given a matrix A of any type, return the same matrix but having as parent the appropriate matrix algebra if A is square, or the appropriate R-matrix space otherwise. This is useful, for example, if it is desired to convert a matrix group element or a square R-matrix space element to be an element of a general matrix algebra.

LowerTriangularMatrix(Q)

Given a sequence Q of length l containing elements of a ring R, such that $l = \binom{n+1}{2} = n(n + 1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ lower-triangular matrix F over R such that the entries of Q describe the lower triangular part of F, in row major order.

LowerTriangularMatrix(R, Q)

Given a ring R and a sequence Q of length l containing elements of a ring S, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ lower-triangular matrix F over R such that the entries of Q, coerced into R, describe the lower triangular part of F, in row major order.

UpperTriangularMatrix(Q)

Given a sequence Q of length l containing elements of a ring R, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ upper-triangular matrix F over R such that the entries of Q describe the upper triangular part of F, in *row major* order.

UpperTriangularMatrix(R, Q)

Given a ring R and a sequence Q of length l containing elements of a ring S, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ upper-triangular matrix F over R such that the entries of Q, coerced into R, describe the upper triangular part of F, in *row major* order.

SymmetricMatrix(Q)

Given a sequence Q of length l containing elements of a ring R, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ symmetric matrix F over R such that the entries of Q describe the lower triangular part of F, in row major order. This function allows the creation of symmetric matrices without the need to specify the redundant upper triangular part.

SymmetricMatrix(R, Q)

Given a ring R and a sequence Q of length l containing elements of a ring S, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ symmetric matrix F over R such that the entries of Q, coerced into R, describe the lower triangular part of F, in row major order. This function allows the creation of symmetric matrices without the need to specify the redundant upper triangular part.

AntisymmetricMatrix(Q)

Given a sequence Q of length l containing elements of a ring R, such that $l = \binom{n}{2} = n(n-1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ antisymmetric matrix F over R such that the entries of Q describe the proper lower triangular part of F, in row major order. The diagonal of F is zero and the proper upper triangular part of F is the negation of the proper lower triangular part of F.

AntisymmetricMatrix(R, Q)

> Given a ring R and a sequence Q of length l containing elements of a ring S, such that $l = \binom{n}{2} = n(n-1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ antisymmetric matrix F over R such that the entries of Q, coerced into R, describe the proper lower triangular part of F, in row major order.

PermutationMatrix(R, Q)

> Given a ring R and a sequence Q of length n, such that Q is a permutation of $[1, 2, \ldots, n]$, return the n by n permutation matrix over R corresponding Q.

PermutationMatrix(R, x)

> Given a ring R and a permutation x of degree n, return the n by n permutation matrix over R corresponding x.

Example H26E3_____

This example demonstrates ways of creating special matrices.
(a) Defining a 3×3 scalar matrix over $\mathbf{Z}$:

```
> S := ScalarMatrix(3, -4);
> S;
[-4  0  0]
[ 0 -4  0]
[ 0  0 -4]
> Parent(S);
Full Matrix Algebra of degree 3 over Integer Ring
```

(b) Defining a 3×3 diagonal matrix over $\mathbf{F}_{23}$:

```
> D := DiagonalMatrix(GF(23), [1, 2, -3]);
> D;
[ 1  0  0]
[ 0  2  0]
[ 0  0 20]
> Parent(D);
Full Matrix Algebra of degree 3 over GF(23)
```

(c) Defining a 3×3 symmetric matrix over $\mathbf{Q}$:

```
> S := SymmetricMatrix([1, 1/2,3, 1,3,4]);
> S;
[  1 1/2   1]
[1/2   3   3]
[  1   3   4]
> Parent(S);
Full Matrix Algebra of degree 3 over Rational Field
```

(d) Defining $n \times n$ lower- and upper-triangular matrices for various n:

```
> low := func<n | LowerTriangularMatrix([i: i in [1 .. Binomial(n + 1, 2)]])>;
```

```
> up := func<n | UpperTriangularMatrix([i: i in [1 .. Binomial(n + 1, 2)]])>;
> sym := func<n | SymmetricMatrix([i: i in [1 .. Binomial(n + 1, 2)]])>;
> low(3);
[1 0 0]
[2 3 0]
[4 5 6]
> up(3);
[1 2 3]
[0 4 5]
[0 0 6]
> sym(3);
[1 2 4]
[2 3 5]
[4 5 6]
> up(6);
[ 1  2  3  4  5  6]
[ 0  7  8  9 10 11]
[ 0  0 12 13 14 15]
[ 0  0  0 16 17 18]
[ 0  0  0  0 19 20]
[ 0  0  0  0  0 21]
```

26.2.4　Construction of Random Matrices

RandomMatrix(R, m, n)

> Given a *finite* ring R and positive integers m and n, construct a random $m \times n$ matrix over R.

RandomUnimodularMatrix(M, n)

> Given positive integers M and n, construct a random integral $n \times n$ matrix having determinant 1 or -1. Most entries will lie in the range $[-M, M]$.

RandomSLnZ(n, k, l)

> A random element of $SL_n(\mathbf{Z})$, obtained by multiplying l random matrices of the form $I + E$, where E has exactly one nonzero entry, which is off the diagonal and has absolute value at most k.

RandomGLnZ(n, k, l)

> A random element of $GL_n(\mathbf{Z})$, obtained in a similar way to RandomSLnZ.

RandomSymplecticMatrix(g, m)

> Given positive integers n and m, construct a (somewhat) random $2n \times 2n$ symplectic matrix over the integers. The entries will have the same order of magnitude as m.

26.2.5   Creating Vectors

Vector(n, Q)

Given an integer n and a sequence Q of length n containing elements of a ring R, return the vector of length n whose entries are the entries of Q. The integer n may be 0, in which case Q must have length 0 and some universe R. This function is equivalent to `RSpace(Universe(Q), n)!Q`.

Vector(Q)

Given a sequence Q of length l containing elements of a ring R, return the vector of length l whose entries are the entries of Q. The argument Q may have length 0 if it has a universe R (i.e., it may not be null). This function is equivalent to `RSpace(Universe(Q), #Q)!Q`.

Vector(R, n, Q)

Given a ring R, an integer n, and a sequence Q of length n containing elements of a ring S, return the vector of length n whose entries are the entries of Q, coerced into R. The integer n may be 0, in which case Q must have length 0 (and may even be null). This function is equivalent to `RSpace(R, n)!Q`.

Vector(R, Q)

Given a ring R and a sequence Q of length l containing elements of a ring S, return the vector of length l whose entries are the entries of Q, coerced into R. The argument Q may have length 0 and may be null. This function is equivalent to `RSpace(R, #Q)!Q`.

26.3   Elementary Properties

The following functions yield elementary properties of matrices and may be applied to matrices of any type, including vectors.

NumberOfRows(A)
Nrows(A)

Given an $m \times n$ matrix A, return m, the number of rows of A.

NumberOfColumns(A)
Ncols(A)

Given an $m \times n$ matrix A, return n, the number of columns of A.

NumberOfNonZeroEntries(A)
NNZEntries(A)

Given a matrix A, return the number of non-zero entries in A.

Density(A)

> Given a matrix A, return the density of A as a real number, which is the number of non-zero entries in A divided by the product of the number of rows of A and the number of columns of A (or zero if A has zero rows or columns).

BaseRing(A)

CoefficientRing(A)

> Given a matrix A with entries lying in a ring R, return R.

ElementToSequence(A)

Eltseq(A)

> Given a matrix A over the ring R having m rows and n columns, return the entries of A, in row-major order, as a sequence of mn elements of R.

RowSequence(A)

> Returns the entries of A as a sequence of rows where a row is represented as a sequence of entries of A.

26.4 Accessing or Modifying Entries

26.4.1 Indexing

The following functions and operators enable one to access individual entries or rows of matrices or vectors.

A[i]

> Given a matrix A over the ring R having m rows and n columns, and an integer i such that $1 \leq i \leq m$, return the i-th row of A, as a vector of length n.

A[i, j]

> Given a matrix A over the ring R having m rows and n columns, integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, return the (i,j)-th entry of A, as an element of the ring R.

A[Q]

A[i .. j]

> Given a matrix A over the ring R having m rows and n columns, and a sequence Q of integers in the range $[1..m]$, return the sequence consisting of the rows of A specified by Q. This is equivalent to [A[i]: i in Q]]. If Q is a range, then the second form A[i .. j] may be used to specify the range directly.

> | `A[i] := v` |

> Given a matrix A over the ring R having m rows and n columns, an integer i such that $1 \le i \le m$, and a vector v over R of length n, modify the i-th row of A to be v. The integer 0 may also be given for v, indicating the zero vector.

> | `A[i, j] := x` |

> Given a matrix A over the ring R having m rows and n columns, integers i and j such that $1 \le i \le m$ and $1 \le j \le n$, and a ring element x coercible into R, modify the (i, j)-th entry of A to be x.

Example H26E4

This example demonstrates simple ways of accessing the entries of matrices.

```
> X := Matrix(4, [1,2,3,4, 5,4,3,2, 1,2,3,4]);
> X;
[1 2 3 4]
[5 4 3 2]
[1 2 3 4]
> X[1];
(1 2 3 4)
> X[1, 2];
2
> X[1, 2] := 23;
> X;
[ 1 23  3  4]
[ 5  4  3  2]
[ 1  2  3  4]
> X[3] := Vector([9,8,7,6]);
> X[2] := 0;
> X;
[ 1 23  3  4]
[ 0  0  0  0]
[ 9  8  7  6]
```

26.4.2 Extracting and Inserting Blocks

The following functions enable the extraction of certain rows, columns or general submatrices, or the replacement of a block by another matrix.

> | `Submatrix(A, i, j, p, q)` |

> | `ExtractBlock(A, i, j, p, q)` |

> Given an $m \times n$ matrix A and integers i, j, p and q such that $1 \le i \le i + p \le m + 1$ and $1 \le j \le j + q \le n + 1$, return the $p \times q$ submatrix of A rooted at (i, j). Either or both of p and q may be zero, while i may be $m + 1$ if p is zero and j may be $n + 1$ if q is zero.

SubmatrixRange(A, i, j, r, s)

ExtractBlockRange(A, i, j, r, s)

> Given an $m \times n$ matrix A and integers i, j, r and s such that $1 \leq i$, $i - 1 \leq r \leq m$, $1 \leq j$, and $j - 1 \leq s \leq n$, return the $r - i + 1 \times s - j + 1$ submatrix of A rooted at the (i, j)-th entry and extending to the (r, s)-th entry, inclusive. r may equal $i - 1$ or s may equal $j - 1$, in which case a matrix with zero rows or zero columns, respectively, will be returned.

Submatrix(A, I, J)

> Given an $m \times n$ matrix A and integer sequences I and J, return the submatrix of A given by the row indices in I and the column indices in J.

InsertBlock(A, B, i, j)

InsertBlock($\sim$A, B, i, j)

> Given an $m \times n$ matrix A over a ring R, a $p \times q$ matrix B over R, and integers i and j such that $1 \leq i \leq i + p \leq m + 1$ and $1 \leq j \leq j + q \leq n + 1$, insert B at position (i, j) in A. In the functional version (A is a value argument), this function returns the new matrix and leaves A untouched, while in the procedural version ($\sim A$ is a reference argument), A is modified in place so that the $p \times q$ submatrix of A rooted at (i, j) is now equal to B.

RowSubmatrix(A, i, k)

> Given an $m \times n$ matrix A and integers i and k such that $1 \leq i \leq i + k \leq m + 1$, return the $k \times n$ submatrix of X consisting of rows $[i \ldots i + k - 1]$ inclusive. The integer k may be zero and i may also be $m + 1$ if k is zero, but the result will always have n columns.

RowSubmatrix(A, i)

> Given an $m \times n$ matrix A and an integer i such that $0 \leq i \leq m$, return the $i \times n$ submatrix of X consisting of the first i rows. The integer i may be 0, but the result will always have n columns.

RowSubmatrixRange(A, i, j)

> Given an $m \times n$ matrix A and integers i and j such that $1 \leq i$ and $i - 1 \leq j \leq m$, return the $j - i + 1 \times n$ submatrix of X consisting of rows $[i \ldots j]$ inclusive. The integer j may equal $i - 1$, in which case a matrix with zero rows and n columns will be returned.

ColumnSubmatrix(A, i, k)

> Given an $m \times n$ matrix A and integers i and k such that $1 \leq i \leq i + k \leq n + 1$, return the $m \times k$ submatrix of X consisting of columns $[i \ldots i + k - 1]$ inclusive. The integer k may be zero and i may also be $n + 1$ if k is zero, but the result will always have m rows.

ColumnSubmatrix(A, i)

> Given an $m \times n$ matrix A and an integer i such that $0 \le i \le n$, return the $m \times i$ submatrix of X consisting of the first i columns. The integer i may be 0, but the result will always have m rows.

ColumnSubmatrixRange(A, i, j)

> Given an $m \times n$ matrix A and integers i and j such that $1 \le i$ and $i - 1 \le j \le n$, return the $m \times j - i + 1$ submatrix of X consisting of columns $[i \ldots j]$ inclusive. The integer j may equal $i - 1$, in which case a matrix with zero columns and n rows will be returned.

Example H26E5_____

The use of the submatrix operations is illustrated by applying them to a 6×6 matrix over the ring of integers $\mathbf{Z}$.

```
> A := Matrix(6,
>     [ 9, 1, 7, -3, 2, -1,
>        3, -4, -5, 9, 2, 7,
>        7, 1, 0, 1, 8, 22,
>        -3, 3, 3, 8, 8, 37,
>        -9, 0, 7, -1, 2, 3,
>        7, 2, -2, 4, 3, 47 ]);
> A;
[ 9  1  7 -3  2 -1]
[ 3 -4 -5  9  2  7]
[ 7  1  0  1  8 22]
[-3  3  3  8  8 37]
[-9  0  7 -1  2  3]
[ 7  2 -2  4  3 47]
> Submatrix(A, 2,2, 3,3);
[-4 -5  9]
[ 1  0  1]
[ 3  3  8]
> SubmatrixRange(A, 2,2, 3,3);
[-4 -5]
[ 1  0]
> S := $1;
> InsertBlock(~A, S, 5,5);
> A;
[ 9  1  7 -3  2 -1]
[ 3 -4 -5  9  2  7]
[ 7  1  0  1  8 22]
[-3  3  3  8  8 37]
[-9  0  7 -1 -4 -5]
[ 7  2 -2  4  1  0]
> RowSubmatrix(A, 5, 2);
```

```
[-9  0  7 -1 -4 -5]
[ 7  2 -2  4  1  0]
> RowSubmatrixRange(A, 2, 3);
[ 3 -4 -5  9  2  7]
[ 7  1  0  1  8 22]
> RowSubmatrix(A, 2, 0);
Matrix with 0 rows and 6 columns
```

26.4.3   Row and Column Operations

The following functions and procedures provide elementary row or column operations on matrices. For each operation, there is a corresponding function which creates a new matrix for the result (leaving the input matrix unchanged), and a corresponding procedure which modifies the input matrix in place.

> SwapRows(A, i, j)

> SwapRows(∼A, i, j)

> > Given an $m \times n$ matrix A and integers i and j such that $1 \le i \le m$ and $1 \le j \le m$, swap the i-th and j-th rows of A.

> SwapColumns(A, i, j)

> SwapColumns(∼A, i, j)

> > Given an $m \times n$ matrix A and integers i and j such that $1 \le i \le n$ and $1 \le j \le n$, swap the i-th and j-th columns of A.

> ReverseRows(A)

> ReverseRows(∼A)

> > Given a matrix A, reverse all the rows of A.

> ReverseColumns(A)

> ReverseColumns(∼A)

> > Given a matrix A, reverse all the columns of A.

> AddRow(A, c, i, j)

> AddRow(∼A, c, i, j)

> > Given an $m \times n$ matrix A over a ring R, a ring element c coercible into R, and integers i and j such that $1 \le i \le m$ and $1 \le j \le m$, add c times row i of A to row j of A.

```
AddColumn(A, c, i, j)
```
```
AddColumn(∼A, c, i, j)
```

Given an $m \times n$ matrix A over a ring R, a ring element c coercible into R, and integers i and j such that $1 \leq i \leq n$ and $1 \leq j \leq n$, add c times column i of A to column j.

```
MultiplyRow(A, c, i)
```
```
MultiplyRow(∼A, c, i)
```

Given an $m \times n$ matrix A over a ring R, a ring element c coercible into R, and an integer i such that $1 \leq i \leq m$, multiply row i of A by c (on the left).

```
MultiplyColumn(A, c, i)
```
```
MultiplyColumn(∼A, c, i)
```

Given an $m \times n$ matrix A over a ring R, a ring element c coercible into R, and an integer i such that $1 \leq i \leq n$, multiply column i of A by c (on the left).

```
RemoveRow(A, i)
```
```
RemoveRow(∼A, i)
```

Given an $m \times n$ matrix A and an integer i such that $1 \leq i \leq m$, remove row i from A (leaving an $(m - 1) \times n$ matrix).

```
RemoveColumn(A, j)
```
```
RemoveColumn(∼A, j)
```

Given an $m \times n$ matrix A and an integer j such that $1 \leq j \leq n$, remove column j from A (leaving an $m \times (n - 1)$ matrix).

```
RemoveRowColumn(A, i, j)
```
```
RemoveRowColumn(∼A, i, j)
```

Given an $m \times n$ matrix A and integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, remove row i and column j from A (leaving an $(m - 1) \times (n - 1)$ matrix).

```
RemoveZeroRows(A)
```
```
RemoveZeroRows(∼A)
```

Given a matrix A, remove all the zero rows of A.

Example H26E6_____

The use of row and column operations is illustrated by applying them to a 5×6 matrix over the ring of integers **Z**.

```
> A := Matrix(5, 6,
>     [ 3, 1, 0, -4, 2, -12,
>       2, -4, -5, 5, 23, 6,
>       8, 0, 0, 1, 5, 12,
>       -2, -6, 3, 8, 9, 17,
>       11, 12, -6, 4, 2, 27 ]);
> A;
[  3   1   0  -4   2 -12]
[  2  -4  -5   5  23   6]
[  8   0   0   1   5  12]
[ -2  -6   3   8   9  17]
[ 11  12  -6   4   2  27]
> SwapColumns(~A, 1, 2);
> A;
[  1   3   0  -4   2 -12]
[ -4   2  -5   5  23   6]
[  0   8   0   1   5  12]
[ -6  -2   3   8   9  17]
[ 12  11  -6   4   2  27]
> AddRow(~A, 4, 1, 2);
> AddRow(~A, 6, 1, 4);
> AddRow(~A, -12, 1, 5);
> A;
[  1   3   0  -4   2 -12]
[  0  14  -5 -11  31 -42]
[  0   8   0   1   5  12]
[  0  16   3 -16  21 -55]
[  0 -25  -6  52 -22 171]
> RemoveRow(~A, 1);
> A;
[ 2 -4 -5  5 23  6]
[ 8  0  0  1  5 12]
[-2 -6  3  8  9 17]
[11 12 -6  4  2 27]
> RemoveRowColumn(~A, 4, 6);
> A;
[ 2 -4 -5  5 23]
[ 8  0  0  1  5]
[-2 -6  3  8  9]
```

26.5 Building Block Matrices

Block matrices can be constructed either by listing the blocks, or by joining together smaller matrices horizontally, vertically or diagonally.

BlockMatrix(m, n, blocks)

> The matrix constructed from the given block matrices, which should all have the same dimensions, and should be given as a sequence of $m \cdot n$ block matrices (given in row major order, in other words listed across rows).

BlockMatrix(m, n, rows)

BlockMatrix(rows)

> The matrix constructed from the given block matrices, which should all have the same dimensions, and should be given as a sequence of m rows, each containing n block matrices.

HorizontalJoin(X, Y)

> Given a matrix X with r rows and c columns, and a matrix Y with r rows and d columns, both over the same coefficient ring R, return the matrix over R with r rows and $(c + d)$ columns obtained by joining X and Y horizontally (placing Y to the right of X).

HorizontalJoin(Q)

HorizontalJoin(T)

> Given a sequence Q or tuple T of matrices, each having the same number of rows and being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q or T horizontally in order.

VerticalJoin(X, Y)

> Given a matrix X with r rows and c columns, and a matrix Y with s rows and c columns, both over the same coefficient ring R, return the matrix with $(r + s)$ rows and c columns over R obtained by joining X and Y vertically (placing Y underneath X).

VerticalJoin(Q)

VerticalJoin(T)

> Given a sequence Q or tuple T of matrices, each having the same number of columns and being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q or T vertically in order.

DiagonalJoin(X, Y)

Given matrices X with a rows and b columns and Y with c rows and d columns, both over the same coefficient ring R, return the matrix with $(a+c)$ rows and $(b+d)$ columns over R obtained by joining X and Y diagonally (placing Y diagonally to the right of and underneath X, with zero blocks above and below the diagonal).

DiagonalJoin(Q)

DiagonalJoin(T)

Given a sequence Q or tuple T of matrices, each being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q or T diagonally in order.

KroneckerProduct(A, B)

Given an $m \times n$ matrix A and a $p \times q$ matrix B, both over a ring R, return the Kronecker product of A and B, which is the $mp \times nq$ matrix C over R such that the $((i-1)p+r, (j-1)q+s)$-th entry of C is the (i,j)-th entry of A times the (r,s)-th entry of B, for $1 \le i \le m$, $1 \le j \le n$, $1 \le r \le p$ and $1 \le s \le q$.

26.6 Changing Ring

ChangeRing(A, R)

Matrix(R, A)

Given a matrix A over a ring S having m rows and n columns, and another ring R, return the $m \times n$ matrix over R obtained by coercing the entries of A from S into R. The argument order to **ChangeRing(A, R)** here is consistent with other forms of **ChangeRing**, while the **Matrix(R, A)** form of this function is provided to be consistent with the matrix creation functions above, for which the destination ring is the first argument, if supplied.

ChangeRing(A, R, f)

ChangeRing(A, f)

Given a matrix A over a ring S having m rows and n columns, another ring R, and a map $f : S \to R$, return the $m \times n$ matrix over R obtained by applying f to each of the entries of A. The R may be omitted, in which case it is taken to be the codomain of f.

CanChangeRing(A, R)

Return whether the entries in the matrix A can be coerced into R and the matrix resulting from **ChangeRing(A, R)** if so.

26.7 Elementary Arithmetic

`A + B`

Given $m \times n$ matrices A and B over a ring R, return $A + B$.

`A - B`

Given $m \times n$ matrices A and B over a ring R, return $A - B$.

`A * B`

Given an $m \times n$ matrix A over a ring R and an $n \times p$ matrix B over R, return the $m \times p$ matrix $A \cdot B$ over R. This function attempts to preserve the maximal amount of information in the choice of parent for the product. For example, if A and B are both square and have the same matrix algebra M as parent, then the product will also have M as parent. Similarly, if the parents of A and B are R-matrix spaces such that the codomain of B equals the domain A, then the product will have domain equal to that of A and codomain equal to that of B.

`x * A`

`A * x`

Given an $m \times n$ matrix A over a ring R and a ring element x coercible into R, return the scalar product $x \cdot A$.

`-A`

Given a matrix A, return $-A$.

`A ^ -1`

Given a invertible square matrix A over a ring R, return the inverse B of A so that $A \cdot B = B \cdot A = 1$. The coefficient ring R must be either a field, a Euclidean domain, or a ring with an exact division algorithm and having characteristic equal to zero or greater than m (this includes most commutative rings).

`A ^ n`

Given a square matrix A over a ring R and an integer n, return the matrix power A^n. A^0 is defined to be the identity matrix for any square matrix A (even if A is zero). If n is negative, A must be invertible (see the previous function), and the result is $(A^{-1})^{-n}$.

`Transpose(A)`

Given an $m \times n$ matrix A over a ring R, return the transpose of A, which is simply the $n \times m$ matrix over R whose (i, j)-th entry is the (j, i)-th entry of A.

`AddScaledMatrix(A, s, B)`

Given a matrix A over a ring R, a scalar s coercible into R, and a matrix B over R with the same shape as A, return $A + s \cdot B$. This is generally quicker than the call `A + s*B`.

AddScaledMatrix(∼A, s, B)

> Given a matrix A over a ring R, a scalar s coercible into R, and a matrix B over R with the same shape as A, set A to $A + s \cdot B$. This is generally quicker than the statement `A := A + s*B;`.

26.8   Nullspaces and Solutions of Systems

The following functions compute nullspaces of matrices (solving equations of the form $V \cdot A = 0$), or solve systems of the form $V \cdot A = W$, for given A and W.

Magma possesses a very rich suite of internal algorithms for computing nullspaces of matrices efficiently, including a fast p-adic algorithm for matrices over $\mathbf{Z}$ and $\mathbf{Q}$, and also algorithms which take advantage of sparsity if it is present.

Nullspace(A)

Kernel(A)

> Given an $m \times n$ matrix A over a ring R, return the nullspace of A (or the kernel of A, considered as a linear transformation or map), which is the R-space consisting of all vectors v of length m such that $v \cdot A = 0$. If the parent of A is an R-matrix space, then the result will be the appropriate submodule of the domain of A. The function `Kernel(A)` also returns the inclusion map from the kernel into the domain of A, to be consistent with other forms of the `Kernel` function.

NullspaceMatrix(A)

KernelMatrix(A)

> Given an $m \times n$ matrix A over a ring R, return a basis matrix of the nullspace of A. This is a matrix N having m columns and the maximal number of independent rows subject to the condition that $N \cdot A = 0$. This function has the advantage that the nullspace is not returned as a R-space, so echelonization of the resulting nullspace may be avoided.

NullspaceOfTranspose(A)

> This function is equivalent to `Nullspace(Transpose(A))`, but may be more efficient in space for large matrices, since the transpose may not have to be explicitly constructed to compute the nullspace.

IsConsistent(A, W)

> Given an $m \times n$ matrix A over a ring R, and a vector W of length n over R or a $r \times n$ matrix W over R, return `true` iff and only if the system of linear equations $V \cdot A = W$ is consistent. If the system is consistent, then the function will also return:
>
> (a) A particular solution V so that $V \cdot A = W$;
>
> (b) The nullspace N of A so that adding any elements of N to any rows of V will yield other solutions to the system.

IsConsistent(A, Q)

> Given an $m \times n$ matrix A over a ring R, and a sequence Q of vectors of length n over R, return `true` if and only if the system of linear equations $V[i] * A = Q[i]$ for all i is consistent. If the system is consistent, then the function will also return:
>
> (a) A particular solution sequence V;
>
> (b) The nullspace N of A so that $(V[i] + u) * A = Q[i]$ for $u \in N$ for all i.

Solution(A, W)

> Given an $m \times n$ matrix A over a ring R, and a vector W of length n over R or a $r \times n$ matrix W over R, solve the system of linear equations $V \cdot A = W$ and return:
>
> (a) A particular solution V so that $V \cdot A = W$;
>
> (b) The nullspace N of A so that adding any elements of N to any rows of V will yield other solutions to the system.
>   If there is no solution, an error results.

Solution(A, Q)

> Given an $m \times n$ matrix A over a ring R, and a sequence Q of vectors of length n over R, solve the system of linear equations $V[i] * A = Q[i]$ for each i and return:
>
> (a) A particular solution sequence V;
>
> (b) The nullspace N of A so that $(V[i] + u) * A = Q[i]$ for $u \in N$ for all i.
>   If there is no solution, an error results.

Example H26E7_____

We compute the nullspace of a 301×300 matrix over $\mathbf{Z}$ with random entries in the range $[0..10]$. The nullity is 1 and the entries of the non-zero null vector are integers, each having about 455 decimal digits.

```
> m := 301; n := 300;
> X := Matrix(n, [Random(0, 10): i in [1 .. m*n]]);
> time N := NullspaceMatrix(X);
Time: 9.519
> Nrows(N), Ncols(N);
1 301
> time IsZero(N*X);
true
Time: 0.429
> {#Sprint(N[1,i]): i in [1..301]};
{ 452, 455, 456, 457, 458 }
```

Example H26E8_____

We show how one can enumerate all solutions to the system $V \cdot X = W$ for a given matrix X and vector W over a finite field. The `Solution` function gives a particular solution for V, and then adding this to every element in the nullspace N of X, we obtain all solutions.

```
> K := GF(3);
> X := Matrix(K, 4, 3, [1,2,1, 2,2,2, 1,1,1, 1,0,1]);
> X;
[1 2 1]
[2 2 2]
[1 1 1]
[1 0 1]
> W := Vector(K, [0,1,0]);
> V, N := Solution(X, W);
> V;
(1 1 0 0)
> N;
Vector space of degree 4, dimension 2 over GF(3)
Echelonized basis:
(1 0 1 1)
(0 1 1 0)
> [V + U: U in N];
[
    (1 1 0 0),
    (2 1 1 1),
    (0 1 2 2),
    (0 2 0 2),
    (1 2 1 0),
    (2 2 2 1),
    (2 0 0 1),
    (0 0 1 2),
    (1 0 2 0)
]
> [(V + U)*X eq W: U in N];
[ true, true, true, true, true, true, true, true, true ]
```

26.9 Predicates

The functions in this section test various properties of matrices. See also the Lattices chapter for a description of the function `IsPositiveDefinite` and related functions.

IsZero(A)

Given an $m \times n$ matrix A over the ring R, return `true` iff A is the $m \times n$ zero matrix.

IsOne(A)

Given a square $m \times m$ matrix A over the ring R, return `true` iff A is the $m \times m$ identity matrix.

IsMinusOne(A)

Given a square $m \times m$ matrix A over the ring R, return `true` iff A is the negation of the $m \times m$ identity matrix.

IsScalar(A)

Given a square $m \times m$ matrix A over the ring R, return `true` iff A is scalar, i.e., iff A is the product of some element of R and the $m \times m$ identity matrix.

IsDiagonal(A)

Given a square matrix A over the ring R, return `true` iff A is diagonal, i.e., iff the only non-zero entries of A are on the diagonal.

IsSymmetric(A)

Given a square matrix A over the ring R, return `true` iff A is symmetric, i.e., iff A equals its transpose.

IsUpperTriangular(A)

Given a matrix A over the ring R, return `true` iff A is upper triangular, i.e., iff the only non-zero entries of A are on or above the diagonal.

IsLowerTriangular(A)

Given a matrix A over the ring R, return `true` iff A is lower triangular, i.e., iff the only non-zero entries of A are on or below the diagonal.

IsUnit(A)

Given a square matrix A over the ring R, return `true` iff A is a unit, i.e., iff A has an inverse. The coefficient ring R may be any commutative ring (since the computation depends on testing if the determinant is a unit – a calculation which is supported in all commutative rings).

IsSingular(A)

Given a square $m \times m$ matrix A over the ring R, return **true** iff A is singular, i.e., iff the determinant of A is zero (or, equivalently, iff the rank of A is less than m). Note that (**not IsSingular(A)**) is *not* equivalent to **IsUnit(A)** whenever R is not a field: if the determinant of A is non-zero but not a unit, then A is non-singular but not invertible. The coefficient ring R may be any commutative ring (since the computation involves only computing the determinant and testing whether it is zero).

IsSymplecticMatrix(A)

Given an $m \times m$ matrix A over the integers, return **true** if and only if A is an integer symplectic matrix, that is, $AJ^tA = J$, where $J = \begin{pmatrix} 0 & \mathbf{1}_g \\ -\mathbf{1}_g & 0 \end{pmatrix}$.

26.10 Determinant and Other Properties

Determinant(A: *parameters*)

MonteCarloLevel	RNGINTELT	*Default* : 0
Proof	BOOLELT	*Default* : **true**
pAdic	BOOLELT	*Default* : **true**
Divisor	RNGINTELT	*Default* : 0

Given a square matrix A over the ring R, return the determinant of A as an element of R. R may be any commutative ring. The determinant of the 0×0 matrix over R is defined to be **R!1**.

If the coefficient ring is the integer ring $\mathbf{Z}$ or the rational field $\mathbf{Q}$ then a modular algorithm based on that of Abbott et al. [ABM99] is used, which first computes a divisor d of the determinant D using a fast p-adic nullspace computation, and then computes the quotient D/d by computing the determinant D modulo enough small primes to cover the Hadamard bound divided by d. This always yields a correct answer.

If the parameter **MonteCarloLevel** is set to a small positive integer s, then a probabilistic Monte-Carlo modular technique is used. Rather than using sufficient primes to cover the Hadamard bound divided by the divisor d, this version of the algorithm terminates when the constructed residue remains constant for s steps. The probability of this being wrong is non-zero but extremely small, even if s is only 1 or 2. If the level is set to 0, then the normal deterministic algorithm is used. Setting the parameter **Proof** to **false** is equivalent to setting **MonteCarloLevel** to 2.

If the coefficient ring is $\mathbf{Z}$ and the parameter **Divisor** is set to an integer d, then d must be a known exact divisor of the determinant (the sign does not matter), and the algorithm may be sped up because of this knowledge.

> **Trace(A)**

Given a square matrix A over the ring R, return the trace of A as an element of R, which is simply the sum of the diagonal elements of A.

> **TraceOfProduct(A, B)**

Given square matrices A and B over the ring R, with the same size, return the trace of $A \cdot B$ as an element of R. This is in general much faster than the call `Trace(A*B)`.

> **Rank(A)**

Given an $m \times n$ matrix A over a ring R, return the rank of A. This is defined to be the largest r such that there exists a non-zero $r \times r$ subdeterminant of A, so $r \le m$ and $r \le n$. The rank may have to be obtained by computing the Smith form or echelon form of A, and this computation may be quite expensive over some rings.

> **Minor(M, i, j)**

The determinant of the submatrix of M (which must be square) formed by removing the i-th row and j-th column.

> **Minor(M, I, J)**

The determinant of the submatrix of M given by the row indices in I and the column indices in J.

> **Minors(M, r)**

Returns a sequence of all the r by r minors of the matrix M.

> **Cofactor(M, i, j)**

The appropriate cofactor of M, equal to $(-1)^{i+j}$ times the corresponding minor.

> **Cofactors(M)**

Returns a sequence of all the cofactors of the matrix M.

> **Cofactors(M, r)**

Returns a sequence of all the r by r cofactors of the matrix M.

> **Pfaffian(M)**
> **Pfaffian(M, I, J)**
> **Pfaffians(M, r)**

Let M be an anti-symmetric square matrix. Then its determinant is always a square and a particular square-root of this, which can be described by a universal polynomial in its entries, is called the Pfaffian of M. The first function returns this. The second function returns the Pfaffian of the submatrix of M described by the indices in I and J. The third function returns the sequence of Pfaffians of the $\binom{n}{r}$ principal r by r submatrices of M ($n =$ the number of rows of M).

These are primarily convenience functions and are computed naively by Pfaffian row-expansion.

26.11   Minimal and Characteristic Polynomials and Eigenvalues

The functions in this section deal with minimal and characteristic polynomials.

MinimalPolynomial(A: *parameters***)**

Proof	BOOLELT	*Default :* true

Given a square matrix A over a ring R, return the minimal polynomial of A. This is defined to be the unique monic univariate polynomial $f(x)$ of minimal degree such that $f(A) = 0$, and $f(x)$ always divides the characteristic polynomial of A. The coefficient ring R is currently restricted to being a field or the integer ring **Z**.

Setting the parameter Proof to false suppresses proof of correctness.

CharacteristicPolynomial(A: *parameters***)**

Al	MONSTG	*Default :* "*Modular*"
Proof	BOOLELT	*Default :* true

Given a square matrix A over a ring R, return the characteristic polynomial of A. This is defined to be the monic univariate polynomial $\mathrm{Det}(x - A) \in R[x]$ where $R[x]$ is the univariate polynomial ring over R. R may be any commutative ring.

The parameter Al allows the user to specify which algorithm that is to be employed. The algorithm "Modular" (the default) may be used for matrices over **Z** and **Q**—in such a case the parameter Proof can also be used to suppress proof of correctness.

The algorithm "Hessenberg", available for matrices over fields, works by first reducing the matrix to Hessenberg form. The algorithm "Interpolation", available for matrices over Z and Q, works by evaluating the characteristic matrix of a at various points and then interpolating. The algorithm "Trace", available for matrices over fields, works by calculating the traces of powers of a.

Since V2.8, none of these algorithms are now recommended for matrices over **Z** or **Q**, as the new p-adic modular algorithm over the integers is extremely fast.

MinimalAndCharacteristicPolynomials(A: *parameters***)**

MCPolynomials(A)

Proof	BOOLELT	*Default :* true

Given a square matrix A over a ring R, return the minimal and characteristic polynomials of A. For some rings, both polynomials can be computed at the same time, so in such cases it will be more efficient to use this function than to call MinimalPolynomial and CharacteristicPolynomials separately.

Setting the parameter Proof to false suppresses proof of correctness.

FactoredMinimalPolynomial(A: *parameters*)

Proof	BOOLELT	*Default :* true

Given a square matrix A over a ring R, return the factorization of the minimal polynomial of A. This is equivalent to `Factorization(MinimalPolynomial(A))`, but may be faster than that for some coefficient rings (in particular, $\mathbf{Z}$ and $\mathbf{Q}$). Setting the parameter `Proof` to false suppresses proof of correctness.

FactoredCharacteristicPolynomial(A: *parameters*)

Al	MONSTG	*Default :* "*Modular*"
Proof	BOOLELT	*Default :* true

Given a square matrix A over a ring R, return the factorisation of the characteristic polynomial of A. This function returns the same result as the command `Factorisation(CharacteristicPolynomial(A))`, but may be faster than that for some coefficient rings (in particular, $\mathbf{Z}$ and $\mathbf{Q}$). The parameters are as for the function `CharacteristicPolynomial` above (setting the parameter `Proof` to false suppresses proof of correctness).

FactoredMinimalAndCharacteristicPolynomials(A: *parameters*)

FactoredMCPolynomials(A: *parameters*)

Al	MONSTG	*Default :* "*Modular*"
Proof	BOOLELT	*Default :* true

Given a square matrix A over a ring R, return the factorizations of the minimal and characteristic polynomials of A, respectively. For some rings, both polynomials can be computed and factored at the same time, so in such cases it will be more efficient to use this function than to call the above functions separately. Setting the parameter `Proof` to false suppresses proof of correctness.

Eigenvalues(A)

Given a square matrix A over a ring R, return the eigenvalues of A as a set of pairs, each of which gives the value of a distinct eigenvalue and its multiplicity. Factorization of polynomials over the base ring R must be possible.

Eigenspace(A, e)

Given a square matrix A over a ring R, and an element e of R, return the eigenspace of A corresponding to e, which is `Nullspace(A - e)`. If the ring element e is not a eigenvalue for the matrix A then the trivial nullspace is returned.

26.12   Canonical Forms

26.12.1   Canonical Forms over General Rings

The functions defined here apply to matrices defined over fields or Euclidean domains. See also the section on Reduction in the Lattices chapter for a description of the function LLL and related basis-reduction functions for matrices.

EchelonForm(A)

> Given an $m \times n$ matrix A over the ring R, return the (reduced) row echelon form E of A, and also an invertible $m \times m$ transformation matrix T over R such that $T \cdot A = E$. Recall that T is a product of elementary matrices that transforms A into the echelon form E. If R is a Euclidean domain, the function HermiteForm (described below) is invoked. Note however, that the the user cannot set the parameters for HermiteForm when invoking it via EchelonForm.

Adjoint(A)

> Given a square $m \times m$ matrix A over the ring R, return the adjoint of A as an $m \times m$ matrix. The base ring R must be a ring with exact division whose characteristic is zero or greater than m (this includes most commutative rings).

26.12.2   Canonical Forms over Fields

The functions described in this section apply to square matrices defined over fields which support factorization of univariate polynomials. See [Ste97] for a description of the single algorithm which is the basis of most of these functions.

PrimaryRationalForm(A)

> Given a square matrix A over a field K such that factorization of polynomials is possible over K, return the primary rational form of A. Each block in the form is the companion matrix of a power of an irreducible polynomial. This function returns three values:
>
> (a) The primary rational canonical form F of A;
>
> (b) An invertible matrix T such that $T \cdot A \cdot T^{-1} = F$;
>
> (c) A sequence of pairs corresponding to the blocks of F where each pair consists of the irreducible polynomial and the multiplicity making up the block. This is the value returned by PrimaryInvariantFactors(A).

JordanForm(A)

> Given a square matrix A over a field K such that factorization of polynomials is possible over K, return the generalized Jordan form of A. Each block in the form is a Jordan block (which itself is derived from a power of an irreducible polynomial), and the generalized Jordan form corresponds to the usual Jordan form if the minimal polynomial splits over K. This function returns three values:
>
> (a) The Jordan canonical form F of A;

(b) An invertible matrix T such that $T \cdot A \cdot T^{-1} = F$;

(c) A sequence of pairs corresponding to the blocks of F where each pair consists of the irreducible polynomial and the multiplicity making up the block. This is the value returned by `PrimaryInvariantFactors(A)`.

`RationalForm(A)`

Given a square matrix A over a field K such that factorization of polynomials is possible over K, return the rational form of A. For each block other than the final block, the polynomial corresponding to that block divides the polynomial corresponding to the next block. This function returns three values:

(a) The rational form F of A;

(b) An invertible matrix T such that $T \cdot A \cdot T^{-1} = F$;

(c) A sequence containing the polynomials corresponding to the successive blocks (where each polynomial, other than the last, divides the next polynomial). This is the value returned by `InvariantFactors(A)`.

`PrimaryInvariantFactors(A)`

Given a square matrix A over a field K such that factorization of polynomials is possible over K, return the primary invariant factors of A. This is the same as the third return value of `PrimaryRationalForm(A)` or `JordanForm(A)`.

`InvariantFactors(A)`

Given a square matrix A over a field K such that factorization of polynomials is possible over K, return the invariant factors of A. This is the same as the third return value of `RationalForm(A)`.

`IsSimilar(A, B)`

Given square $m \times m$ matrices A and B, both over a field K such that factorization of polynomials is possible over K, return `true` iff and only if A is similar to B, and if so, return also an invertible $m \times m$ transformation matrix T such that $T \cdot A \cdot T^{-1} = B$.

`HessenbergForm(A)`

Given a square $m \times m$ matrix A over the ring R, return the Hessenberg form of A as an $m \times m$ matrix. The form has zero entries above the super-diagonal. (This form is used in one of the characteristic polynomial algorithms.) The base ring R must be a field.

`FrobeniusFormAlternating(A)`

Given a non-singular $2n \times 2n$ alternating matrix A over the integers, this function returns the (alternating) Frobenius form F of A. That is, a block matrix $F = \begin{pmatrix} 0 & D \\ -D & 0 \end{pmatrix}$, where D is a diagonal matrix with positive diagonal entries, d_i, satisfying $d_1 | d_2 | \cdots | d_n$. The second return value is the change of basis matrix B, such that $BA^t B = F$.

Example H26E9_____

We construct a 5×5 matrix over the finite field with 5 elements and then calculate various canonical forms. We verify the correctness of the polynomial invariant factors corresponding to the rational form by calculating the Smith form of the characteristic matrix of the original matrix (see below).

```
> K := GF(5);
> A := Matrix(K, 5,
>     [ 0, 2, 4, 2, 0,
>       2, 2, 2, 3, 3,
>       3, 4, 4, 1, 3,
>       0, 0, 0, 0, 1,
>       0, 0, 0, 1, 0 ]);
> A;
[0 2 4 2 0]
[2 2 2 3 3]
[3 4 4 1 3]
[0 0 0 0 1]
[0 0 0 1 0]
> PrimaryInvariantFactors(A);
[
    <x + 1, 1>,
    <x + 1, 1>,
    <x + 4, 1>,
    <x + 4, 1>,
    <x + 4, 1>
]
> JordanForm(A);
[4 0 0 0 0]
[0 4 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
> R, T, F := RationalForm(A);
> R;
[1 0 0 0 0]
[0 0 1 0 0]
[0 1 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
> T;
[1 3 0 2 1]
[2 1 2 2 0]
[3 4 3 4 1]
[1 0 0 0 0]
[0 2 4 2 0]
> T*A*T^-1 eq R;
true;
```

```
> F;
[
    x + 4,
    x^2 + 4,
    x^2 + 4
]
> P<x> := PolynomialRing(K);
> PM := MatrixAlgebra(P, 5);
> Ax := PM ! x - PM ! A;
> Ax;
[    x      3      1      3      0]
[  3 x + 3      3      2      2]
[    2      1 x + 1      4      2]
[    0      0      0      x      4]
[    0      0      0      4      x]
> SmithForm(Ax);
[      1        0        0        0        0]
[      0        1        0        0        0]
[      0        0    x + 4        0        0]
[      0        0        0 x^2 + 4        0]
[      0        0        0        0 x^2 + 4]
> ElementaryDivisors(Ax);
[
    1,
    1,
    x + 4,
    x^2 + 4,
    x^2 + 4
]
```

26.12.3   Canonical Forms over Euclidean Domains

The functions defined here apply to matrices defined over Euclidean domains. See also the section on Reduction in the Lattices chapter for a description of the function LLL and related functions, which are very useful for integer matrices.

HermiteForm(A)		
Al	MONSTG	*Default :* "*Default*"
Optimize	BOOLELT	*Default :* true
Integral	BOOLELT	*Default :* true

Given an $m \times n$ matrix A over the Euclidean ring R, return the Hermite form H of A, and also an invertible $m \times m$ transformation matrix T over R such that $T \cdot A = H$.

The basic algorithm used is the classical Kannan-Bachem algorithm [KB79, CC82], which has classical complexity (but does not suffer from bad coefficient growth).

Since V2.13, for matrices over the integers there is also a fast modular algorithm by Allan Steel. By default, MAGMA chooses between these two algorithms, usually favouring the new modular algorithm. But one may set the parameter `Al` to `"Modular"` to force the modular algorithm to be used, and to `"Classical"` to force the classical algorithm to be used.

If R is the ring of integers $\mathbf{Z}$ and the matrix T is requested (i.e., if an assignment statement is used with two variables on the left side), then the LLL algorithm will also be used by default to improve T (using the kernel of A) so that the size of its entries are very small. If the parameter `Optimize` is set to `false`, then this will not happen (which will be faster but the entries of T will not be as small). If the parameter `Integral` is set to `true`, then the integral (de Weger) LLL method will be used in the LLL step, instead of the default floating point method.

SmithForm(A)

Given an $m \times n$ matrix A over the Euclidean ring R, return the Smith normal form of A. This function returns three values:

(a) The Smith normal form S of A; and

(b) Unimodular matrices P and Q such that $P \cdot A \cdot Q = S$, i.e., P and Q are matrices which transform A into Smith normal form.

The algorithm implemented first uses the sparse techniques described in [HHR93] to reduce the matrix to a dense submatrix, then, if this is non-trivial, it either repeatedly calls the Hermite normal form algorithm (see above) and transposes until a diagonal form is obtained, or uses the modular algorithm of F. Lübeck [Lüb02].

Unless one wishes one or both of the transformation matrices, it is preferable to use the following function `ElementaryDivisors` since it gives the same information, but saves memory since the matrix S does not need to be constructed.

ElementaryDivisors(A)

Given an $m \times n$ matrix A over the Euclidean ring or field R, return the elementary divisors of A. These are simply the non-zero diagonal entries of the Smith form of A, in order. The divisors are returned as a sequence $[e_1, \ldots, e_r]$ of r elements of R (which may include ones), where r is the rank of A and $e_i | e_{i+1}$ for $i = 1, \ldots, r-1$. The divisors are normalized, so the result is unique. If R is a field, the result is always the sequence of r ones, where r is the rank of A.

Note that if $m = n = r$, then the determinant of A is the product of the e_i and if R is also a domain, then e_r is the lowest common denominator of the inverse of A over the field of fractions of R.

Saturation(A)

Given an $m \times n$ matrix A over the integer ring $\mathbf{Z}$, having rank r, return an $m \times n$ matrix S over $\mathbf{Z}$ whose first r rows form a basis of the saturation w.r.t. $\mathbf{Q}$ of the $\mathbf{Q}$-vector space spanned by the rows of A. The rows of S thus span the same space over $\mathbf{Q}$ as those of A, while the $\mathbf{Z}$-module spanned by the rows of S is the set of all

v such that for some non-zero scalar s, $s \cdot v$ is in the **Z**-module spanned by the rows of A.

Example H26E10_____

We illustrate some of these operations for a 4×3 matrix over $\mathbf{F}_8$.

```
> K<w> := GF(8);
> A := Matrix(K, 4, 3, [1,w,w^5, 0,w^3,w^4, w,1,w^6, w^3,1,w^4]);
> A;
[  1   w w^5]
[  0 w^3 w^4]
[  w   1 w^6]
[w^3   1 w^4]
> EchelonForm(A);
[  1   0   0]
[  0   1   0]
[  0   0   1]
[  0   0   0]
```

We now illustrate some of these operations for a 4×5 matrix over **Z**.

```
> A := Matrix(4, 5,
>     [ 2,-4,12,7,0,
>        3,-3,5,-1,4,
>        2,-1,-4,-5,-12,
>        0,3,6,-2,0]);
> A;
[  2  -4  12   7   0]
[  3  -3   5  -1   4]
[  2  -1  -4  -5 -12]
[  0   3   6  -2   0]
> Rank(A);
4
> HermiteForm(A);
[   1    1    1    6 -164]
[   0    3    0   16 -348]
[   0    0    2   13 -200]
[   0    0    0   19 -316]
> SmithForm(A);
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 2 0]
> ElementaryDivisors(A);
[ 1, 1, 1, 2 ]
```

26.13   Orders of Invertible Matrices

The functions defined here apply to invertible square matrices. MAGMA can efficiently compute the order of an invertible matrix over a finite field, using the Cunningham database to factorize the numbers of the form $p^n - 1$ which arise. The algorithm employed is that described in [CLG97a].

MAGMA also contains efficient algorithms for rigorously proving whether a matrix over the ring of integers $\mathbf{Z}$, the rational field $\mathbf{Q}$, an algebraic number field, a cyclotomic field or a quadratic field has finite order or not, and for determining the order if it is finite.

HasFiniteOrder(A)

> Given a square invertible matrix A over a ring R, return true iff A has finite order, i.e., iff there exists a positive integer n such that $A^n = 1$. The coefficient ring R is currently restricted to being either a finite field, the ring of integers $\mathbf{Z}$, the rational field $\mathbf{Q}$, an algebraic number field, a cyclotomic field or a quadratic field. For matrices over any of these rings, the function rigorously proves its result (over other rings, an error results).

Order(A)

   Proof   BOOLELT   *Default* : true

> Given a square invertible matrix A over any commutative ring, return the order of A. If R is a ring for which a finite order proof exists (see HasFiniteOrder above), then an error results if A has infinite order. Over other rings, if A has infinite order then the function may loop indefinitely since it may not be able to prove the infinitude of the order.

FactoredOrder(A)

   Proof   BOOLELT   *Default* : true

> Given a square invertible matrix A over a finite field, return the order of A in factored form. This returns the same value as Factorization(Order(A)), but since the order computation must compute the factorization of the order anyway, it involves no more effort to have it return the factorization. The conditions on the ring are as for Order.

ProjectiveOrder(A)

   Proof   BOOLELT   *Default* : true

> Given a square invertible matrix A over a finite field K, return the projective order n of A and a scalar $s \in K$ such that $A^n = sI$. The projective order of A is the smallest n such that A^n is a scalar matrix (not just the identity matrix), and it always divides the true order of A. The parameter Proof is as for Order.

FactoredProjectiveOrder(A)

Proof   BOOLELT   *Default :* true

> Given a square invertible matrix A over a finite field K, return the projective order
> n of A in factored form and a scalar $s \in K$ such that $A^n = sI$. The parameter
> Proof is as for FactoredOrder.

26.14   Numerical Linear Algebra

MAGMA has some limited capabilities with numerical linear algebra, that is, linear algebra
over a real or complex field. The basis of these is the RQDecomposition (appropriate as
MAGMA is row-based), and its companion QLDecomposition. Internally, MAGMA uses
these functions to compute (e.g.) matrix inverses and determinants over **R** and **C**.

All of the intrinsics starting with Numeric can also be accessed with this word omitted
(by extended types on a real/complex matrix), though for some purposes it is useful to
remind oneself that numerical linear algebra is not always the same.

RQDecomposition(M)

> Given a matrix M over a real or complex field, write $M = RQ$ where R is of
> triangular form and Q orthogonal (or unitary in the complex case). In particular,
> R has the same dimensions as M; when M has more columns c than rows r, then
> R is of the form $[Z|U]$ where Z is a zero matrix that has $(c-r)$ columns and U is
> upper triangular. When $r > c$ then the bottom c rows of R form an upper triangular
> matrix. No normalisations are made except that Q has determinant 1.

QLDecomposition(M)

> Given a matrix M over a real or complex field, write $M = QL$ where L is of
> triangular form and Q orthogonal (or unitary in the complex case). This function
> works simply by applying RQDecomposition to the transpose.

NumericalInverse(M)

Epsilon   FLDREELT   *Default :*

> Given a square positive-dimension matrix over a real or complex field, compute its
> inverse. The Epsilon parameter allows one to specify what size of values should
> be considered to be zero (resulting in an inversion failure). By default, the value of
> Epsilon is computed in a way that is relative to the size of the entries and their
> precision.

Example H26E11_____

We give some examples of these decompositions.

```
> RR := RealField(10);
> r := 4;
> c := 5;
> A := [RR!Random([-2^10..2^10])/2^9 : i in [1..r*c]];
```

```
> M := Matrix(r,c,A);
> M;
[-0.1796875000 -1.394531250 -1.345703125 -0.4414062500 0.0000000000]
[1.396484375 -1.496093750 -0.2617187500 -1.882812500 -1.679687500]
[-1.169921875 -0.9062500000 1.707031250 -0.6582031250 1.480468750]
[0.8867187500 -0.3359375000 0.5839843750 0.9707031250 1.896484375]
> R,Q := RQDecomposition(M); R, Q;
[0.0000000000 1.767478576 0.8447303787 -0.05853296288 0.3765439489]
[0.0000000000 0.0000000000 2.924272555 -0.2048205303 1.424771353]
[0.0000000000 0.0000000000 0.0000000000 2.589389831 -1.011949267]
[0.0000000000 0.0000000000 0.0000000000 0.0000000000 -2.403971597]
[-0.1278237197 -0.4893124575 0.3070157953 0.6619247514 -0.4602513884]
[-0.3369945746 -0.5416048144 -0.7236111740 0.05012569151 0.2587917417]
[0.6155218516 -0.6003864917 0.06838441636 -0.4759773097 -0.1715752984]
[-0.5959647961 -0.2953736282 0.5643042371 -0.4119965068 0.2634387749]
[-0.3688557516 0.1397427076 -0.2429248231 -0.4037914284 -0.7888963320]
> Max([Abs(x) : x in Eltseq(R*Q-M)]);
3.492459655E-10 10
> Max([Abs(x) : x in Eltseq(Transpose(Q)*Q-1)]);
1.527951099E-10 14
> //
> CC<i> := ComplexField(5);
> r := 5;
> c := 3;
> A := [CC!Random([-2^10..2^10])/2^9 : j in [1..r*c]];
> B := [CC!Random([-2^10..2^10])/2^9 : j in [1..r*c]];
> M := Matrix(r,c,A) + i*Matrix(r,c,B);
> R,Q := RQDecomposition(M); R, Q;
[-1.1471 + 1.0971*i       -2.6950 - 0.38148*i   0.47294 - 0.95428*i]
[-0.25971 - 0.0084722*i  0.61157 + 0.49147*i    1.5717 - 0.61096*i]
[0.30902 + 0.59626*i      0.79478 + 1.0406*i     -1.1157 - 1.6800*i]
[0.00000                  1.2132 + 1.4143*i     -0.95447 + 0.14362*i]
[0.00000                  0.00000                3.0078 - 0.24966*i]
[0.42713 - 0.0038340*i   0.51414 - 0.56570*i     -0.39550 + 0.27712*i]
[0.54343 + 0.60175*i     -0.087580 + 0.0038354*i  0.56113 + 0.14178*i]
[0.33511 + 0.21874*i     -0.62188 + 0.14580*i     -0.65721]
> Max([Abs(x) : x in Eltseq(R*Q-M)]);
0.00020472 3
> Max([Abs(x) : x in Eltseq(ConjugateTranspose(Q)*Q-1)]);
0.00012207 9
> //
> r := 5;
> c := 5;
> A := [RR!Random([-2^10..2^10])/2^9 : j in [1..r*c]];
> M := Matrix(r,c,A);
> MI := NumericalInverse(M); MI;
[0.5488038571 -0.3734925483 0.005386589048 0.4034292318 -0.6410457874]
[0.3082898339 -0.5126616665 0.1612965318 -0.2971166681 0.07521900340]
```

```
[-0.4446389269 0.4962564367 0.1565165202 -0.4722494193 0.2177560581]
[-0.05815013416 -0.3372341073 0.03559373548 0.5402886617 -0.7996494168]
[-0.9780538200 1.225635935 -0.4735623930 -1.060353816 1.356842221]
> Max([Abs(x) : x in Eltseq(M*MI-1)]);
4.656612873E-10 4
```

26.14.1   Rank, Kernel, Solution, and Pseudoinverse

To compute the kernel (thus also rank) of a matrix M or a solution to a system $vM = w$, the implementation first performs a QL decomposition internally. When L is of a strict triangular form, namely $L = \left(\frac{Z}{A|T} \right)$ where Z is zero and T is not only lower triangular but also invertible, then the information can be read off immediately. In particular, the rank r is the dimension of T, the kernel is the first k rows of Q where k is number of rows in Z, while $vM = w$ requires $vQL = w$ so $v = w_r T^{-1} Q_r^T$ where w_r and Q_r are the rightmost r columns of w and Q; one checks $vQ_r A$ is (numerically) equal to the other columns of w.

When L is not of this form, then an arbitrary orthogonal/unitary transformation is performed on the input (on the right), and the algorithm is re-tried. This case occurs when the r rightmost columns of M are not linearly independent (MAGMA is internally working on the transpose of M in the RQ decomposition, so that the rightmost columns become the lowermost rows). Similarly, if a condition number for T is too large, such a randomising transform can be profitably applied.

Note that the kernel is returned as a matrix, and indeed has orthonormal rows. The same is true for the image.

The computation of the `NumericalPseudoinverse` is similar, being based on the `QLDecomposition` and possibly also transforming the columns on the right, to ensure that L is of the nice triangular form (this is internally called `QLCDecomposition`). The pseudoinverse of the row-independent (bottom) part B of L is then computed as the solution $\tilde{P}$ of $\tilde{P} B B^\star = B^\star$, and one row-extends $\tilde{P}$ to the left (by zero columns) to get the full (numerical) pseudoinverse P.

NumericalRank(M)

Epsilon	FLDREELT	*Default :*

> Given a matrix over a real or complex field, compute its (numerical) rank. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

NumericalKernel(M)

Epsilon	FLDREELT	*Default :*

> Given a matrix over a real or complex field, compute its (numerical) kernel. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

Exceptionally, `Kernel` returns a space, while `NumericalKernel` returns a basis matrix of it.

`NumericalImage(M)`

Epsilon   FldReElt   *Default :*

Given a matrix over a real or complex field, compute its (numerical) image, that is, its row space. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

Exceptionally, `Image` returns a space, while `NumericalImage` returns a basis matrix of it.

`NumericalSolution(M,w)`

Epsilon   FldReElt   *Default :*

Given a matrix M and a compatible matrix (or vector) w over a real or complex field, solve numerically for $vM = w$. This gives an error if there is no solution. The second return value is the kernel of M. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

Exceptionally, the second return value is a matrix for `NumericalSolution` and a space for `Solution`.

`NumericalIsConsistent(M,w)`

Epsilon   FldReElt   *Default :*

Given a matrix M and a compatible matrix (or vector) w over a real or complex field, determine whether $vM = w$ is numerically solvable. The second return value is a solution (when one exists)(, and the third return value is the (numerical) kernel of M. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision.

Exceptionally, the third return value is a matrix for `NumericalIsConsistent` and a space for `IsConsistent`.

`NumericalPseudoinverse(M)`

Epsilon   FldReElt   *Default :*

Given a matrix M over a real or complex field, compute its pseudoinverse P such that $PMP = P$, $MPM = M$, $(MP)^\star = MP$ and $(PM)^\star = PM$. The `Epsilon` parameter allows one to specify what size of values should be considered to be zero. By default, the value of `Epsilon` is computed in a way that is relative to the size of the entries and their precision. Note that the Magma intrinsic `PseudoInverse` is quite different (largely applicable for matrices over Euclidean domains).

Example H26E12_____

We give some examples for ranks, kernels, solutions, and pseudoinverses.

```
> RR := RealField(10);
> r := 3;
> c := 5;
> S := [-2^10..2^10];
> A := [Vector([RR!Random(S)/2^9 : i in [1..c]]) : j in [1..r]];
> M := Matrix(A cat [A[1]+A[2]-A[3]]);
> M;
[-1.347656250 -1.980468750 0.2070312500 0.2734375000 -1.273437500]
[0.6464843750 1.726562500 -0.3769531250 0.2011718750 -1.646484375]
[-0.1660156250 1.236328125 -1.500000000 1.119140625 -1.195312500]
[-0.5351562500 -1.490234375 1.330078125 -0.6445312500 -1.724609375]
> NumericalRank(M);
3
> NumericalKernel(M); // scaled to norm 1
[-0.5000000001 -0.4999999998 0.4999999999 0.5000000002]
> v := Vector([RR!Random(S)/2^9 : i in [1..c]]);
> NumericalIsConsistent(M,v);
false
> w := &+[M[i] * Random(S)/2^9 : i in [1..r]];
> NumericalIsConsistent(M,w);
true (1.174316406 0.4165039062 0.7807617187 0.8100585938)
> PI := NumericalPseudoinverse(M); PI;
[-0.1991067080 0.1335911770 -0.1240987264 0.05858319545]
[-0.1885126384 0.1881125824 0.02268111094 -0.02308116681]
[-0.1331315985 0.1012317424 -0.2422989451 0.2103990890]
[0.1423299946 -0.08451596130 0.1982404871 -0.1404264538]
[-0.07615077666 -0.2393032832 -0.09708672579 -0.2183673341]
> Max([Abs(x) :  x in Eltseq(PI*M*PI-PI)]);
7.275957614E-11 9
> //
> CC<i> := ComplexField(30);
> r := 5;
> c := 4;
> A := [CC!Random([-2^30..2^30])/2^28 : j in [1..r*c]];
> B := [CC!Random([-2^30..2^30])/2^28 : j in [1..r*c]];
> M := Matrix(r,c,A) + i*Matrix(r,c,B);
> NumericalRank(M);
4
> NumericalKernel(M);
[0.208965559769595845191678654168 + 0.0334373308308175861091994295635*i
 -0.0227045638063900267279802631268 - 0.354081499548224953341852663885*i
 -0.220008165304096296179634856382 - 0.815082824683314746916890536889*i
 0.0146505331723745379341299824850 - 0.155641405430086914216739561265*i
 0.209604845631814749141329584762 - 0.219520964338492435702429703559*i]
> PI := NumericalPseudoinverse(M);
```

```
> Max([Abs(x) :  x in Eltseq(PI*M*PI-PI)]);
1.60521564392120817260990102530E-30 1
> Max([Abs(x) :  x in Eltseq(M*PI*M-M)]);
2.54399997817916409183980874314E-29 18
> Max([Abs(x) :  x in Eltseq(M*PI-ConjugateTranspose(M*PI))]);
7.88860905221011805411728565283E-30 25
> Max([Abs(x) :  x in Eltseq(PI*M-ConjugateTranspose(PI*M))]);
5.03961120551682881427899748034E-30 8
```

26.14.2   Eigenvalues and the Singular Value Decomposition

Computing the eigenvalues of a real or complex square matrix is done by first computing the Hessenberg form (almost upper-triangular, but with possible nonzero entries on the immediate subdiagonal), and then applying iterative methods to achieve the Schur form (upper-triangular except for 2-by-2 blocks with complex eigenvalues in the real case). Special methods are used in the case where the matrix is symmetric. The computation of the singular value decomposition works by first computing the bidiagonal form, and then similarly iterating.

NumericalHessenbergForm(M)

> Given a square matrix M over a real/complex field, compute a Hessenberg matrix H that is upper-triangular except for possible nonzero entries on the immediate subdiagonal, and an orthogonal/unitary transformation matrix Q such that $H = QMQ^\star$ and $QQ^\star = I$ (both equalities being numerically approximate). A procedure similar to the RQ decomposition is used, working on both rows and columns.

NumericalSchurForm(M)

Transform   BOOLELT   *Default* : true

> Given a square matrix M over a real/complex field, compute the Schur form S that is upper-triangular except for possible nonzero entries on the immediate subdiagonal corresponding to 2-by-2 real blocks with complex eigenvalues, and an orthogonal/unitary transformation matrix Q such that $S = QMQ^\star$ and $QQ^\star = I$ (both equalities being numerically approximate). First the Hessenberg form is calculated, and then the unwanted subdiagonal terms are removed by a iterative process.

NumericalEigenvalues(M)

Balance	BOOLELT	*Default* : `true`

Given a square matrix M over a real/complex field, compute numerical approximations to its eigenvalues. This operates by calculating the Schur form, and then returning the relevant eigenvalues. A form of Parlett-Reinsch balancing is used by default, to increase the precision of the answer.

Exceptionally, `NumericalEigenvalues` returns a sequence of reals or complexes, while `Eigenvalues` on real/complex matrix returns a tuple of eigenvalues and multiplicities, for reasons of backwards compatibility (the multiplicities really cannot be guaranteed in the numerical case). Note that `NumericalEigenvalues` always returns a full sequence of eigenvalues for a real input, while `Eigenvalues` will return only those which are (approximately) real.

NumericalEigenvectors(M, e)

Given a square matrix M that is coercible into the complexes and an approximation e to an eigenvalue of it, attempt to find eigenvectors. This function uses `NumericalKernel`, though returns the result as a sequence of (row) vectors.

NumericalBidiagonalForm(M)

Given a matrix M over a real/complex field, compute a bidiagonal form B and orthogonal/unitary transformations U, V such that $B = UMV^{\star}$. Here B is upper bidiagonal if the number of columns is at least as large as the number of rows, and else is lower bidiagonal.

NumericalSingularValueDecomposition(M)

Given a matrix M over a real/complex field, compute the singular value decomposition S and orthogonal/unitary transformations U, V such that $S = UMV^{\star}$. Here S is diagonal in its upper-left corner, with the diagonal entries being nonnegative and sorted in increasing size.

Example H26E13_____

We give some examples for eigenvalues.

```
> RR := RealField(10);
> r := 5;
> c := 5;
> A := [RR!Random([-2^10..2^10])/2^9 : i in [1..r*c]];
> M := Matrix(r,c,A);
> H := NumericalHessenbergForm(M); H;
[-1.387758248 0.2244522681 0.1050196798 -0.6292371533 1.500649611]
[-1.737800256 -0.6560018402 -0.1292093616 -0.4045870847 -0.1501588456]
[-0.0000000000 3.503611888 -0.4200132582 -1.521474700 -0.4771033714]
[-0.0000000000 0.0000000000 1.241929396 -1.887789155 -1.398580785]
[-0.0000000000 0.0000000000 0.0000000000 2.063240612 -1.882812500]
```

```
> S := NumericalSchurForm(M); S; // need not be diag in real case
[-0.5276818045 2.970316193 -0.8183434354 -1.227178569 1.207265137]
[-0.6560286404 0.1374330601 0.7474073924 -0.7405697671 0.4496312767]
[0.0000000000 0.0000000000 -2.733003988 0.5326130259 -0.2389927500]
[0.0000000000 0.0000000000 0.0000000000 -1.878704188 2.041990652]
[0.0000000000 0.0000000000 0.0000000000 -2.574033133 -1.232418078]
> NumericalEigenvalues(M);
[ -2.733003988,
  -1.555561133 - 2.269742311*$.1, -1.555561133 + 2.269742311*$.1,
  -0.1951243722 - 1.355735243*$.1, -0.1951243722 + 1.355735243*$.1 ]
> M := M + Transpose(M); // make symmetric
> NumericalEigenvalues(M); // all real
[ -5.927297871, -4.230690093, -3.248785626, -1.661515789, 2.599539376 ]
> //
> // example with companion matrix of a polynomial
> //
> f:=Polynomial([Random(10^(12-j)) : j in [0..8]] cat [1]); f;
x^9 + 8569*x^8 + 98753*x^7 + 533140*x^6 + 3583980*x^5 + 52066941*x^4 +
    183889636*x^3 + 5743452355*x^2 + 22923098485*x + 725907276071
> NumericalEigenvalues(ChangeRing(CompanionMatrix(f),RR));
[ -8557.467295,
  -10.76902934 - 4.819656295*$.1, -10.76902934 + 4.819656295*$.1,
  -4.602845057 - 8.104590519*$.1, -4.602845057 + 8.104590519*$.1,
   2.007741523 - 8.923992632*$.1, 2.007741523 + 8.923992632*$.1,
   7.597780247 - 5.110282521*$.1, 7.597780247 + 5.110282521*$.1 ]
> Sort([r[1] : r in Roots(f,ComplexField(10))],func<x,y|Real(x)-Real(y)>);
[ -8557.467295,
  -10.76902933 - 4.819656325*$.1, -10.76902933 + 4.819656325*$.1,
  -4.602845059 - 8.104590521*$.1, -4.602845059 + 8.104590521*$.1,
   2.007741527 - 8.923992640*$.1, 2.007741527 + 8.923992640*$.1,
   7.597780251 - 5.110282532*$.1, 7.597780251 + 5.110282532*$.1 ]
```

Example H26E14_____

We give some examples for bidiagonal form and singular value decomposition.

```
> CC<i> := ComplexField(5);
> r := 4;
> c := 5;
> A := [CC!Random([-2^10..2^10])/2^9 : i in [1..r*c]];
> B := [CC!Random([-2^10..2^10])/2^9 : i in [1..r*c]];
> M := Matrix(r,c,A) + i*Matrix(r,c,B);
> X := NumericalBidiagonalForm(M); X;
> NumericalBidiagonalForm(M); // matrix over CC
[2.1983 3.7789 0.00000 0.00000 0.00000]
[0.00000 2.0496 1.8677 0.00000 0.00000]
[0.00000 0.00000 2.6879 2.4315 0.00000]
[0.00000 0.00000 0.00000 3.5035 0.00000]
```

```
> NumericalSingularValueDecomposition(M);
[4.9265 0.00000 0.00000 0.00000 0.00000]
[0.00000 4.5508 0.00000 0.00000 0.00000]
[0.00000 0.00000 2.5876 0.00000 0.00000]
[0.00000 0.00000 0.00000 0.73134 0.00000]
> [Sqrt(Real(x)) : x in NumericalEigenvalues(M*ConjugateTranspose(M))];
[ 0.73122, 2.5876, 4.5506, 4.9265 ]
```

26.15 Miscellaneous Operations on Matrices

FrobeniusImage(A, e)

> Given a matrix A over a finite field K of characteristic p, return the matrix obtained from A by mapping each entry $A_{i,j}$ to $(A_{i,j})^{p^e}$.

26.16 Bibliography

[**ABM99**] John Abbott, Manuel Bronstein, and Thom Mulders. Fast Deterministic Computation of Determinants of Dense Matrices. In Sam Dooley, editor, *Proceedings ISSAC'99*, pages 197–204, New York, 1999. ACM Press.

[**CC82**] T.W.J. Chou and G.E. Collins. Algorithms for the solution of systems of linear Diophantine equations. *SIAM J. Computing*, 11(4):687–708, 1982.

[**CLG97**] Frank Celler and Charles R. Leedham-Green. Calculating the Order of an Invertible Matrix. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 55–60. AMS, 1997.

[**HHR93**] George Havas, Derek F. Holt, and Sarah Rees. Recognizing badly presented Z-modules. *Linear Algebra and its Applications*, 192:137–164, 1993.

[**KB79**] R. Kannan and A. Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM J. Computing*, 9:499–507, 1979.

[**Lüb02**] F. Lübeck. On the computation of elementary divisors of integer matrices. *J. Symbolic Comp.*, 33:57–65, 2002.

[**Ste97**] Allan Steel. A New Algorithm for the Computation of Canonical Forms of Matrices over Fields. *J. Symbolic Comp.*, 24(3):409–432, 1997.

27 SPARSE MATRICES

Chapter 27

SPARSE MATRICES

27.1 Introduction

A separate type is provided for sparse matrices to allow the user to construct such matrices and apply algorithms which take advantage of sparsity. Sparse matrices are distinct from normal matrices in MAGMA, which have a dense representation. This chapter describes the operations available for creating and working with sparse matrices.

The operations provided for sparse matrices include dynamic construction, simple properties, and the calculation of a number of non-trivial and important invariants (such as rank, determinant, or computing a non-zero vector in the nullspace). In particular, this datatype supports the class of *index-calculus* algorithms which involve generating a very large sparse system and then solving the system or finding the elementary divisors of the corresponding matrix (for example, to compute the abelian group structure). An extended example presented at the end of the chapter (H27E3) illustrates how an index-calculus method may be implemented in practice in the MAGMA language using the sparse matrix facilities.

The type name for the category of sparse matrices is `MtrxSprs`. All sparse matrices over a given ring R lie in the same *sparse matrix structure*, whose type name is `MtrxSprsStr`. The user will, in practice, rarely need to refer explicitly to the parent structure.

27.2 Creation of Sparse Matrices

This section describes the constructs provided for creating sparse matrices.

27.2.1 Construction of Initialized Sparse Matrices

| SparseMatrix(R, m, n, Q) |

| SparseMatrix(m, n, Q) |

> Given a ring R (optional), integers $m, n \geq 0$ and a sequence Q, return the $m \times n$ sparse matrix over R whose non-zero entries are those specified by Q, coerced into R. If R is not given, it is derived from the entries in Q. Either of m and n may be 0, in which case Q must have length 0 (and may be null if R is given), and the $m \times n$ zero sparse matrix over R is returned. There are two possibilities for Q:
>
> (a) The sequence Q is a sequence of tuples, each of the form `<i, j, x>`, where $1 \leq i \leq m$, $1 \leq j \leq n$, and $x \in S$ for some ring S. Such a tuple specifies that the (i, j)-th entry of the matrix is x. If R is given, then x is coerced into R; otherwise the matrix is created over S. If an entry position is not given then its value is zero, while if an entry position is repeated then the last value overrides any previous values.

(b) The sequence Q is a "flat" sequence of integers, giving the entries of the matrix in compact form. To be precise, Q begins with the number of non-zero entries n for the first row, then $2 \cdot n$ integers giving column-entry pairs for the first row, and this format is immediately followed for the second row and so on. A zero row is specified by a zero value for n. If R is given, the integer entries are coerced into R; otherwise the matrix is defined over $\mathbf{Z}$. (Thus this method will not allow elements of R which cannot be created by coercing integers into R alone; another way of saying this is that the entries must all lie in the prime ring of R). This allows a very compact way to create (and store) sparse matrices. The examples below illustrate this format.

SparseMatrix(R, m, n)

Given a ring R, and integers $m, n \geq 0$, create the $m \times n$ sparse matrix over R.

SparseMatrix(m, n)

Given integers $m, n \geq 0$, create the $m \times n$ sparse matrix over the integer ring $\mathbf{Z}$.

27.2.2 Construction of Trivial Sparse Matrices

SparseMatrix(R)

SparseMatrix()

Create the 0×0 sparse matrix over R. If R is omitted (so there are no arguments), then R is taken to be the integer ring $\mathbf{Z}$. These functions will usually be called when the user wishes to create a sparse matrix whose final dimensions are initially unknown, and for which the SetEntry procedure below will be used to extend the matrix automatically, as needed.

Example H27E1_____

This example demonstrates simple ways of creating matrices using the general SparseMatrix(R, m, n, Q) function. Sparse matrices may be displayed in the sparse representation using the Magma print-format. Also, the function Matrix (described below) takes a sparse matrix and returns a normal (dense-representation) matrix, so this function provides a means of printing a small sparse matrix as a normal matrix.

(a) A sparse 2×2 matrix is defined over $\mathbf{Z}$, using the first (sequence of tuples) method. We specify that there is a 3 in the $(1, 2)$ position and a -1 in the $(2, 3)$ position. The ring need not be given since the entries are in $\mathbf{Z}$ already.

```
> A := SparseMatrix(2, 3, [<1,2,3>, <2,3,-1>]);
> A;
Sparse matrix with 2 rows and 3 columns over Integer Ring
> Matrix(A);
[ 0  3  0]
```

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

(b) The same matrix is now defined over $\mathbf{F}_{23}$. We could also coerce the 3rd component of each tuple into $\mathbf{F}_{23}$ and thus omit the first argument.

```
> A := SparseMatrix(GF(23), 2, 3, [<1,2,3>, <2,3,-1>]);
> A;
Sparse matrix with 2 rows and 3 columns over GF(23)
> Matrix(A);
[ 0  3  0]
[ 0  0 22]
```

(c) A similar sparse matrix is defined over $\mathbf{F}_{2^4}$. When A is printed in `Magma` format, the sequence-of-tuples form is used (because the entries cannot be printed as integers).

```
> K<w> := GF(2^4);
> A := SparseMatrix(K, 2, 3, [<1,2,3>, <2,3,w>]);
> A;
Sparse matrix with 2 rows and 3 columns over GF(2^4)
> Matrix(A);
[   0   1   0]
[   0   0   w]
> A: Magma;
SparseMatrix(GF(2, 4), 2, 3, [
    <1, 2, 1>,
    <2, 3, w>
])
```

(d) A sparse 4×5 matrix A is defined over $\mathbf{Z}$, using the second (flat integer sequence) method. Here row 1 has one non-zero entry: -1 at column 3; then row 2 has three non-zero entries: 9 at column 2, 7 at column 3, and -3 at column 4; row 3 has no non-zero entries (so we give a 0 at this point); and finally row 4 has one non-zero entry 3 at column 4. Note that when A is printed in `Magma` format, this time the compact "flat" sequence of integers form is used, since this is possible.

```
> A := SparseMatrix(4,5, [1,3,-1, 3,2,9,3,7,4,-3, 0, 1,4,3]);
> A;
Sparse matrix with 4 rows and 5 columns over Integer Ring
> Matrix(A);
[ 0  0 -1  0  0]
[ 0  9  7 -3  0]
[ 0  0  0  0  0]
[ 0  0  0  3  0]
> A: Magma;
SparseMatrix(4, 5, \[
    1, 3,-1,
    3, 2,9, 3,7, 4,-3,
    0,
    1, 4,3
])
```

27.2.3 Construction of Structured Matrices

IdentitySparseMatrix(R, n)

Given a ring R, and an integer $n \geq 0$, return the $n \times n$ identity sparse matrix over R.

ScalarSparseMatrix(n, s)

Given an integer $n \geq 0$ and an element s of a ring R, return the $n \times n$ scalar sparse matrix over R which has s on the diagonal and zeros elsewhere. The argument n may be 0, in which case the 0×0 sparse matrix over R is returned.

ScalarSparseMatrix(R, n, s)

Given a ring R, an integer $n \geq 0$ and an element s of a ring S, return the $n \times n$ scalar sparse matrix over R which has s, coerced into R, on the diagonal and zeros elsewhere. n may be 0, in which case in which case the 0×0 sparse matrix over R is returned.

DiagonalSparseMatrix(R, n, Q)

Given a ring R, an integer $n \geq 0$ and a sequence Q of n ring elements, return the $n \times n$ diagonal sparse matrix over R whose diagonal entries correspond to the entries of Q, coerced into R.

DiagonalSparseMatrix(R, Q)

Given a ring R and a sequence Q of n ring elements, return the $n \times n$ diagonal sparse matrix over R whose diagonal entries correspond to the entries of Q, coerced into R.

DiagonalSparseMatrix(Q)

Given a sequence Q of n elements from a ring R, return the $n \times n$ diagonal sparse matrix over R whose diagonal entries correspond to the entries of Q.

27.2.4 Parents of Sparse Matrices

SparseMatrixStructure(R)

Create the structure containing all sparse matrices (of any shape) over ring R. This structure does not need to be created explicitly by the user (it will be the parent of any sparse matrix over R), but it may be useful to create it in this way occasionally.

27.3   Accessing Sparse Matrices

The following functions access basic properties of sparse matrices.

27.3.1   Elementary Properties

`BaseRing(A)`

`CoefficientRing(A)`

> Given a sparse matrix A with entries lying in a ring R, return R.

`NumberOfRows(A)`

`Nrows(A)`

> Given an $m \times n$ sparse matrix A, return m, the number of rows of A.

`NumberOfColumns(A)`

`Ncols(A)`

> Given an $m \times n$ sparse matrix A, return n, the number of columns of A.

`ElementToSequence(A)`

`Eltseq(A)`

> Given a sparse matrix A over the ring R having m rows and n columns, return the entries of A as a sequence of all tuples of the form `<i, j, x>` such that the $[i,j]$-th entry of A equals x and x is non-zero. It is always true that `SparseMatrix(Nrows(A), Ncols(A), Eltseq(A))` equals A.

`NumberOfNonZeroEntries(A)`

`NNZEntries(A)`

> Given a sparse matrix A, return the number of non-zero entries in A.

`Density(A)`

> Given a sparse matrix A, return the density of A as a real number, which is the number of non-zero entries in A divided by the product of the number of rows of A and the number of columns of A (or zero if A has zero rows or columns).

`Support(A, i)`

> Given a sparse matrix A having r rows, and an integer i such that $1 \le i \le r$, return the support of row i of A; i.e., the column numbers of the non-zero entries of row i of A.

`Support(A)`

> Given a sparse matrix A, return the sequence of all pairs `<i, j>` such that the $[i,j]$-th entry of A is non-zero.

27.3.2   Weights

RowWeight(A, i)

> Given a sparse matrix A with m rows and an integer i such that $1 \leq i \leq m$, return the weight (number of non-zero entries) of the i-th row of A.

RowWeights(A)

> Given a sparse matrix A with m rows, return the length m sequence of integers whose i-th entry is the weight of the i-th row of A.

ColumnWeight(A, j)

> Given a sparse matrix A with n columns and an integer j such that $1 \leq j \leq n$, return the weight (number of non-zero entries) of the j-th column of A.

ColumnWeights(A)

> Given a sparse matrix A with n columns, return the length n sequence of integers whose j-th entry is the weight of the j-th column of A.

27.4   Accessing or Modifying Entries

The following functions and procedures enable the user to access or set individual entries of sparse matrices.

A[i]

> Given a sparse matrix A over a ring R having m rows and n columns, and an integer i such that $1 \leq i \leq m$, return the i-th row of A, as a dense vector of length n (lying in R^n).

A[i, j]

> Given a sparse matrix A over a ring R having m rows and n columns, integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, return the (i, j)-th entry of A, as an element of the ring R.

A[i, j] := x

> Given a sparse matrix A over a ring R having m rows and n columns, integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, and a ring element x coercible into R, modify the (i, j)-th entry of A to be x. Here i and j must be within the ranges given by the current dimensions of A; see SetEntry below for a procedure to automatically extend A if necessary.

SetEntry($\sim$A, i, j, x)

    (Procedure.) Given a sparse matrix A over a ring R, integers $i, j \geq 1$, and a ring element x coercible into R, modify the (i, j)-th entry of A to be x. The entry specified by i and j is allowed to be beyond the current dimensions of A; if so, A is automatically extended to have at least i rows and j columns.

    This procedure will be commonly used in situations where the final size of the matrix is not known as an algorithm proceeds (e.g., in index-calculus methods). One can create the 0×0 sparse matrix over $\mathbf{Z}$, say, and then call `SetEntry` to build up the matrix dynamically. See the example H27E3 below, which uses this technique.

    Note that extending the dimensions of A with a very large i or j will not in itself consume much memory, but if A then becomes dense or is passed to some algorithm, then the memory needed may of course be proportional to the dimensions of A.

Example H27E2_____

This example demonstrates simple ways of accessing the entries of sparse matrices.

```
> A := SparseMatrix(2, 3, [<1,2,3>, <2,3,-1>]);
> A;
Sparse matrix with 2 rows and 3 columns over Integer Ring
> Matrix(A);
[ 0  3  0]
[ 0  0 -1]
> A[1];
(0 3 0)
> A[1, 3]:=5;
> A[1];
(0 3 5)
```

We next extend A using the procedure `SetEntry`.

```
> SetEntry(~A, 1, 5, -7);
> A;
Sparse matrix with 2 rows and 5 columns over Integer Ring
> Matrix(A);
[ 0  3  5  0 -7]
[ 0  0 -1  0  0]
```

A common situation is to start with the empty 0×0 matrix over $\mathbf{Z}$ and then to extend it dynamically.

```
> A := SparseMatrix();
> A;
Sparse matrix with 0 rows and 0 columns over Integer Ring
> SetEntry(~A, 1, 4, -2);
> A;
Sparse matrix with 1 row and 4 columns over Integer Ring
> SetEntry(~A, 2, 3, 8);
> A;
```

```
Sparse matrix with 2 rows and 4 columns over Integer Ring
> Matrix(A);
[ 0   0   0 -2]
[ 0   0   8  0]
> SetEntry(~A, 200, 319, 1);
> SetEntry(~A, 200, 3876, 1);
> A;
Sparse matrix with 200 rows and 3876 columns over Integer Ring
> Nrows(A);
200
> Ncols(A);
3876
> NNZEntries(A);
4
> Density(A);
0.00000515995872033023735810113519O
> Support(A, 200);
[ 319, 3876 ]
```

27.4.1   Extracting and Inserting Blocks

The following functions enable the extraction of certain rows, columns or general submatrices, or the replacement of a block by another sparse matrix.

Submatrix(A, i, j, p, q)

ExtractBlock(A, i, j, p, q)

> Given an $m \times n$ sparse matrix A and integers i, j, p and q such that $1 \le i \le i + p \le m + 1$ and $1 \le j \le j + q \le n + 1$, return the $p \times q$ submatrix of A rooted at (i, j). Either or both of p and q may be zero, while i may be $m + 1$ if p is zero and j may be $n + 1$ if q is zero.

SubmatrixRange(A, i, j, r, s)

ExtractBlockRange(A, i, j, r, s)

> Given an $m \times n$ sparse matrix A and integers i, j, r and s such that $1 \le i$, $i - 1 \le r \le m$, $1 \le j$, and $j - 1 \le s \le n$, return the $r - i + 1 \times s - j + 1$ submatrix of A rooted at the (i, j)-th entry and extending to the (r, s)-th entry, inclusive. r may equal $i - 1$ or s may equal $j - 1$, in which case a sparse matrix with zero rows or zero columns, respectively, will be returned.

Submatrix(A, I, J)

> Given an $m \times n$ sparse matrix A and integer sequences I and J, return the submatrix of A given by the row indices in I and the column indices in J.

InsertBlock(A, B, i, j)

InsertBlock($\sim$A, B, i, j)

> Given an $m \times n$ sparse matrix A over a ring R, a $p \times q$ sparse matrix B over R, and integers i and j such that $1 \leq i \leq i + p \leq m + 1$ and $1 \leq j \leq j + q \leq n + 1$, insert B at position (i, j) in A. In the functional version (A is a value argument), this function returns the new sparse matrix and leaves A untouched, while in the procedural version ($\sim A$ is a reference argument), A is modified in place so that the $p \times q$ submatrix of A rooted at (i, j) is now equal to B.

RowSubmatrix(A, i, k)

> Given an $m \times n$ sparse matrix A and integers i and k such that $1 \leq i \leq i + k \leq m + 1$, return the $k \times n$ submatrix of X consisting of rows $[i \ldots i + k - 1]$ inclusive. The integer k may be zero and i may also be $m + 1$ if k is zero, but the result will always have n columns.

RowSubmatrix(A, i)

> Given an $m \times n$ sparse matrix A and an integer i such that $0 \leq i \leq m$, return the $i \times n$ submatrix of X consisting of the first i rows. The integer i may be 0, but the result will always have n columns.

RowSubmatrixRange(A, i, j)

> Given an $m \times n$ sparse matrix A and integers i and j such that $1 \leq i$ and $i - 1 \leq j \leq m$, return the $j - i + 1 \times n$ submatrix of X consisting of rows $[i \ldots j]$ inclusive. The integer j may equal $i - 1$, in which case a sparse matrix with zero rows and n columns will be returned.

ColumnSubmatrix(A, i, k)

> Given an $m \times n$ sparse matrix A and integers i and k such that $1 \leq i \leq i + k \leq n + 1$, return the $m \times k$ submatrix of X consisting of columns $[i \ldots i + k - 1]$ inclusive. The integer k may be zero and i may also be $n + 1$ if k is zero, but the result will always have m rows.

ColumnSubmatrix(A, i)

> Given an $m \times n$ sparse matrix A and an integer i such that $0 \leq i \leq n$, return the $m \times i$ submatrix of X consisting of the first i columns. The integer i may be 0, but the result will always have m rows.

ColumnSubmatrixRange(A, i, j)

> Given an $m \times n$ sparse matrix A and integers i and j such that $1 \leq i$ and $i - 1 \leq j \leq n$, return the $m \times j - i + 1$ submatrix of X consisting of columns $[i \ldots j]$ inclusive. The integer j may equal $i - 1$, in which case a sparse matrix with zero columns and n rows will be returned.

27.4.2   Row and Column Operations

The following functions and procedures provide elementary row or column operations on sparse matrices. For each operation, there is a corresponding function which creates a new sparse matrix for the result (leaving the input sparse matrix unchanged), and a corresponding procedure which modifies the input sparse matrix in place.

> SwapRows(A, i, j)

> SwapRows($\sim$A, i, j)

> Given an $m \times n$ sparse matrix A and integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq m$, swap the i-th and j-th rows of A.

> SwapColumns(A, i, j)

> SwapColumns($\sim$A, i, j)

> Given an $m \times n$ sparse matrix A and integers i and j such that $1 \leq i \leq n$ and $1 \leq j \leq n$, swap the i-th and j-th columns of A.

> ReverseRows(A)

> ReverseRows($\sim$A)

> Given a sparse matrix A, reverse all the rows of A.

> ReverseColumns(A)

> ReverseColumns($\sim$A)

> Given a sparse matrix A, reverse all the columns of A.

> AddRow(A, c, i, j)

> AddRow($\sim$A, c, i, j)

> Given an $m \times n$ sparse matrix A over a ring R, a ring element c coercible into R, and integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq m$, add c times row i of A to row j of A.

> AddColumn(A, c, i, j)

> AddColumn($\sim$A, c, i, j)

> Given an $m \times n$ sparse matrix A over a ring R, a ring element c coercible into R, and integers i and j such that $1 \leq i \leq n$ and $1 \leq j \leq n$, add c times column i of A to column j.

> MultiplyRow(A, c, i)

> MultiplyRow($\sim$A, c, i)

> Given an $m \times n$ sparse matrix A over a ring R, a ring element c coercible into R, and an integer i such that $1 \leq i \leq m$, multiply row i of A by c (on the left).

```
MultiplyColumn(A, c, i)
```
```
MultiplyColumn(∼A, c, i)
```

> Given an $m \times n$ sparse matrix A over a ring R, a ring element c coercible into R, and an integer i such that $1 \leq i \leq n$, multiply column i of A by c (on the left).

```
RemoveRow(A, i)
```
```
RemoveRow(∼A, i)
```

> Given an $m \times n$ sparse matrix A and an integer i such that $1 \leq i \leq m$, remove row i from A (leaving an $(m-1) \times n$ sparse matrix).

```
RemoveColumn(A, j)
```
```
RemoveColumn(∼A, j)
```

> Given an $m \times n$ sparse matrix A and an integer j such that $1 \leq j \leq n$, remove column j from A (leaving an $m \times (n-1)$ sparse matrix).

```
RemoveRowColumn(A, i, j)
```
```
RemoveRowColumn(∼A, i, j)
```

> Given an $m \times n$ sparse matrix A and integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, remove row i and column j from A (leaving an $(m-1) \times (n-1)$ sparse matrix).

```
RemoveZeroRows(A)
```
```
RemoveZeroRows(∼A)
```

> Given a sparse matrix A, remove all the zero rows of A.

27.5   Building Block Matrices

```
HorizontalJoin(A, B)
```

> Given a sparse matrix A with r rows and c columns, and a sparse matrix B with r rows and d columns, both over the same coefficient ring R, return the sparse matrix over R with r rows and $(c+d)$ columns obtained by joining A and B horizontally (placing B to the right of A).

```
VerticalJoin(A, B)
```

> Given a sparse matrix A with r rows and c columns, and a sparse matrix B with s rows and c columns, both over the same coefficient ring R, return the sparse matrix with $(r+s)$ rows and c columns over R obtained by joining A and B vertically (placing B underneath A).

DiagonalJoin(A, B)

> Given matrices A with a rows and b columns and B with c rows and d columns, both over the same coefficient ring R, return the sparse matrix with $(a + c)$ rows and $(b + d)$ columns over R obtained by joining A and B diagonally (placing B diagonally to the right of and underneath A, with zero blocks above and below the diagonal).

27.6   Conversion to and from Dense Matrices

The following functions convert between sparse matrices and normal (dense-representation) matrices.

Matrix(A)

> Given a sparse matrix A, return the normal (dense-representation) matrix equal to A. This function should only be used if the size of A is reasonably small since otherwise the amount of memory needed to represent the dense-representation matrix may be huge. Printing the result of this operation is a convenient way to display A as a normal (dense) matrix if A is small.

SparseMatrix(A)

> Given a normal (dense-representation) matrix A, return the sparse matrix equal to A. Note that if there is a fast algorithm available for the sparse matrix type, there is no need to convert a dense-representation matrix to the sparse matrix type first and then to use that algorithm, since MAGMA does this automatically.

27.7   Changing Ring

ChangeRing(A, R)

SparseMatrix(R, A)

> Given a sparse matrix A over a ring S having m rows and n columns, and another ring R, return the $m \times n$ sparse matrix over R obtained by coercing the entries of A from S into R.
>
> The argument order to ChangeRing(A, R) here is consistent with other forms of ChangeRing, while the SparseMatrix(R, A) form of this function is provided to be consistent with the sparse matrix creation functions above, for which the destination ring is the first argument, if supplied.

27.8 Predicates

The functions in this section test various properties of sparse matrices.

A eq B

> Given sparse matrices A and B, return **true** if and only if A and B are equal.

IsZero(A)

> Given an $m \times n$ sparse matrix A over the ring R, return **true** iff A is the $m \times n$ zero sparse matrix.

IsOne(A)

> Given a square $m \times m$ sparse matrix A over the ring R, return **true** iff A is the $m \times m$ identity sparse matrix.

IsMinusOne(A)

> Given a square $m \times m$ sparse matrix A over the ring R, return **true** iff A is the negation of the $m \times m$ identity sparse matrix.

IsScalar(A)

> Given a square $m \times m$ sparse matrix A over the ring R, return **true** iff A is scalar, i.e., iff A is the product of some element of R and the $m \times m$ identity sparse matrix.

IsDiagonal(A)

> Given a square sparse matrix A over the ring R, return **true** iff A is diagonal, i.e., iff the only non-zero entries of A are on the diagonal.

IsSymmetric(A)

> Given a square sparse matrix A over the ring R, return **true** iff A is symmetric, i.e., iff A equals its transpose.

IsUpperTriangular(A)

> Given a sparse matrix A over the ring R, return **true** iff A is upper triangular, i.e., iff the only non-zero entries of A are on or above the diagonal.

IsLowerTriangular(A)

> Given a sparse matrix A over the ring R, return **true** iff A is lower triangular, i.e., iff the only non-zero entries of A are on or below the diagonal.

27.9   Elementary Arithmetic

| A + B |

Given $m \times n$ sparse matrices A and B over a ring R, return $A + B$.

| A – B |

Given $m \times n$ sparse matrices A and B over a ring R, return $A - B$.

| A * B |

Given an $m \times n$ sparse matrix A over a ring R and an $n \times p$ sparse matrix B over R, return the $m \times p$ sparse matrix $A \cdot B$ over R.

| x * A |
| A * x |

Given an $m \times n$ sparse matrix A over a ring R and a ring element x coercible into R, return the scalar product $x \cdot A$.

| –A |

Given a sparse matrix A, return $-A$.

| A ^-1 |

Given a invertible square sparse matrix A over a ring R, return the inverse B of A so that $A \cdot B = B \cdot A = 1$. The coefficient ring R must be either a field, a Euclidean domain, or a ring with an exact division algorithm and having characteristic equal to zero or greater than m (this includes most commutative rings).

| A ^ n |

Given a square sparse matrix A over a ring R and an integer n, return the matrix power A^n. A^0 is defined to be the identity matrix for any square matrix A (even if A is zero). If n is negative, A must be invertible (see the previous function), and the result is $(A^{-1})^{-n}$.

| Transpose(A) |

Given an $m \times n$ sparse matrix A over a ring R, return the transpose of A, which is simply the $n \times m$ sparse matrix over R whose (i, j)-th entry is the (j, i)-th entry of A.

27.10   Multiplying Vectors or Matrices by Sparse Matrices

These functions allow the multiplication of a normal (dense-representation) vector by a sparse matrix.

v * A

V * A

> Given a dense-representation vector v or dense-representation matrix V with c columns, together with a sparse $c \times n$ matrix A, both over a ring R, return the product $v \cdot A$ or $V \cdot A$.
>
> This is generally fast if A is sparse and uses minimal memory.

MultiplyByTranspose(v, A)

MultiplyByTranspose(V, A)

> Given a dense-representation vector v or dense-representation matrix V with c columns, together with a sparse $n \times c$ matrix A, both over a ring R, return the product of v or V by the transpose of A.
>
> This is generally fast if A is sparse, and is much faster than computing the transpose of A first. For example, if the vector-matrix product $v \cdot A \cdot A^{tr}$ is required, then the function call MultiplyByTranspose(v*A, A) should be used to avoid forming the matrix $A \cdot A^{tr}$ which is usually dense. This product occurs in iterative algorithms such as Lanzcos.

27.11   Non-trivial Properties

The following functions compute non-trivial properties of sparse matrices.

27.11.1   Nullspace and Rowspace

The following functions compute nullspaces (solving equations of the form $V \cdot A = 0$) or rowspaces of sparse matrices.

Nullspace(A)

Kernel(A)

> Given an $m \times n$ sparse matrix A over a ring R, return the nullspace of A (or the kernel of A, considered as a linear transformation or map), which is the R-space consisting of all vectors v of length m such that $v \cdot A = 0$. Since the result will be given in the dense representation, both the nullity of A and the number of rows of A must both be reasonably small.
>
> The algorithm first performs sparse elimination using Markowitz pivoting ([DEJ84, Sec. 9.2]) to obtain a smaller dense matrix, then the nullspace algorithm for dense-representation matrices is applied to this matrix.

NullspaceMatrix(A)

KernelMatrix(A)

Given an $m \times n$ sparse matrix A over a ring R, return a (dense-representation) basis matrix of the nullspace of A. This is a matrix N having m columns and the maximal number of independent rows subject to the condition that $N \cdot A = 0$. This function has the advantage that the nullspace is not returned as a R-space, so echelonization of the resulting nullspace may be avoided.

NullspaceOfTranspose(A)

This function is equivalent to `Nullspace(Transpose(A))`, but will be more efficient in space for large matrices, since the transpose may not have to be explicitly constructed to compute the nullspace.

Rowspace(A)

Given an $m \times n$ sparse matrix A over a ring R, return the rowspace of A, which is the R-space generated by the rows of A. Since the result will be given in the dense representation, the rank and the number of columns of A must both be reasonably small.

27.11.2   Rank

Rank(A)

Given an $m \times n$ sparse matrix A over a ring R, return the rank of A. The algorithm first performs sparse elimination using Markowitz pivoting ([DEJ84, Sec. 9.2]) to obtain a smaller dense matrix, then the rank algorithm for dense-representation matrices is applied to this matrix.

27.12   Determinant and Other Properties

Determinant(A: *parameters***)**

  MonteCarloSteps   Rng Int Elt   *Default :*

Given a square sparse matrix A over the ring R, return the determinant of A as an element of R. R may be any commutative ring.

The algorithm first performs sparse elimination using Markowitz pivoting ([DEJ84, Sec. 9.2]) to obtain a smaller dense matrix, then the determinant algorithm for dense-representation matrices is applied to this matrix. If the parameter `MonteCarloSteps` is given, then this is passed to the dense algorithm for the dense matrix.

27.12.1 Elementary Divisors (Smith Form)

ElementaryDivisors(A)

Given an $m \times n$ matrix A over the Euclidean ring or field R, return the elementary divisors of A. These are simply the non-zero diagonal entries of the Smith form of A, in order.

The divisors are returned as a sequence $Q = [e_1, \ldots, e_d]$, $e_i | e_{i+1}$ $(i = 1, \ldots, d-1)$ of d elements of R (which may include ones), where d is the rank of A. If R is a field, the result is always a sequence of r ones, where r is the rank of A.

A function for computing the Smith normal form is not supplied for sparse matrices since the form may be trivially derived from the elementary divisors, and the sequence Q containing the divisors is often more convenient (and takes less memory). As transformation matrices are dense in general, they are not supported for the sparse representation.

The algorithm first performs sparse elimination using Markowitz pivoting to obtain a smaller dense matrix ([DEJ84, Sec. 9.2]; this is similar to the techniques described in [HHR93]). Then it invokes the dense Smith normal form algorithm for normal (dense-representation) matrices (`SmithForm`).

27.12.2 Verbosity

SetVerbose("SparseMatrix", v)

(Procedure.) Set the verbose printing level for all sparse matrix algorithms to be v. Currently the legal values for v are `true`, `false`, 0, 1, 2, and 3 (`false` has the same effect as 0, and `true` has the same effect as 1).

27.13 Linear Systems (Structured Gaussian Elimination)

ModularSolution(A, M)

ModularSolution(A, L)

Lanczos   BoolElt   *Default* : `false`

Given a sparse $m \times n$ matrix A, defined over the integer ring $\mathbf{Z}$, and a positive integer M, compute a vector v such that v satisfies the equation $v \cdot A^{tr} = 0$ modulo M. v will be non-zero with high probability.

This function is designed for index-calculus-type algorithms where a large sparse linear system defined by the matrix A is first constructed and then a vector satisfying the above equation modulo M is required. For this reason it is natural that the transpose of A appears in this equation. The example H27E3 below illustrates such a situation in detail.

The first version of the function takes the actual integer M as the second argument given and so must be factored as part of the calculation, while the second

version of the function takes the factorization sequence L of M. If possible, the solution vector is multiplied by a unit modulo M so that its first entry is 1.

The function uses the *Structured Gaussian Elimination* algorithm [LO91b, Sec. 5]. This reduces the linear system to be solved to a much smaller but denser system. By default, the function recurses on the smaller system until it is almost completely dense, and then this dense system is solved using the fast dense modular nullspace algorithm of MAGMA.

If the parameter `Lanczos` is set to `true`, then the *Lanczos* algorithm [LO91b, Sec. 3] will be used instead. This is generally *very much* slower than the default method (it is often 10 to 50 times slower), but it will take considerably less memory, so may be preferable in the case of extremely large matrices.

For typical matrices arising in index-calculus problems, and for most machines, the default method (reducing to a completely dense system) should solve a linear system of size roughly $100,000 \times 100,000$ using about 500MB of memory while a linear system of size roughly $200,000 \times 200,000$ should require about 1.5GB to 2.0GB of memory.

Example H27E3 _____

In this extended example, we demonstrate the application of the function `ModularSolution` to a sparse matrix arising in an index-calculus algorithm. We present MAGMA code which performs the first stage of the basic linear sieve [COS86, LO91a] for computing discrete logarithms in a prime finite field $\mathbf{F}_p$. This first stage determines most of the logarithms of the elements of the *factor basis* (which is the set of small primes up to a given limit) by using a *sieve* to compute a sparse linear system which is then solved modulo $p - 1$.

Even though the following sieving code is written in the MAGMA language and so is *much* slower than a serious C implementation, it is sufficiently powerful to be able to compute the first stage logarithms in less than a minute for fields $\mathbf{F}_p$ where p is about 10^{20} and $(p-1)/2$ is prime. In comparison, the standard Pohlig-Hellman algorithm based on the Pollard-Rho method would take many hours for such fields. This MAGMA code can also be adapted for other index-calculus methods.

Suppose p is an odd prime and let $K = \mathbf{F}_p$. Let Q be the *factor base*, an ordered set consisting of all positive primes from 2 to a given limit `qlimit`. Fix a primitive element α of K which is also *prime* as an integer, so α is in Q. We wish to compute the discrete logarithms of most of the elements of Q with respect to α; i.e., we wish to compute l_q with $\alpha^{l_q} = q$ for most $q \in Q$.

The algorithm uses a *sieve* to search for linear relations involving the logarithms of the elements of Q. Let $H = \lfloor \sqrt{p} \rfloor + 1$ and $J = H^2 - p$. We search for pairs of integers (c_1, c_2) with $1 \le c_1, c_2 \le$ `climit` (where `climit` is a given limit which is much less than H) such that

$$[(H + c_1)(H + c_2)] \bmod p = J + (c_1 + c_2)H + c_1 c_2$$

is *smooth* with respect to Q (i.e., all of its prime factors are in Q). If we include these $H + c_i$ in the factor base, then this gives a linear relation modulo $(p - 1)$ among the logarithms of the elements of the extended factor base.

Fix c_1 with $1 \leq c \leq$ `climit` and suppose we initialize a sieve array (to be indexed by c_2) to have zero in each position. For each prime power q^h with $q \in Q$ and h sufficiently small, we compute

$$d = [(J + c_1 H)(H + c_1)^{-1}] \bmod q^h.$$

Then for all $c_2 \equiv d \pmod{q^h}$, it turns out that

$$(H + c_1)(H + c_2) \equiv 0 \pmod{q^h}.$$

So for each c_2 with $c_1 \leq c_2 \leq$ `climit` and $c_2 \equiv d \pmod{q^h}$, we add $\log(q)$ to the position of the sieve corresponding to c_2. (Ensuring that $c_2 \geq c_1$ avoids redundant relations.) When we have done this for each $q \in Q$, we perform trial division to obtain relations for each of the c_2 whose sieve values are beneath a suitable threshold.

We repeat this with a new c_1 value until we have more relations than elements of the factor base (typically we make the ratio be 1.1 or 1.2), then we solve the corresponding linear system modulo $M = p - 1$ to obtain the desired logarithms. We ensure that α is the first element of Q so that when the vector is normalized modulo M (so that its first entry is 1), the logarithms will be with respect to α. For derivations of the above equations and for further details concerning the sieving, see [LO91a].

In practice, one first writes $M = p - 1 = M_1 \cdot M_2$ where M_1 contains the maximal powers of all the small primes dividing M (say, for primes ≤ 10000). The solution space modulo M_1 will often have high dimension, so the logarithms modulo M_1 usually cannot be correctly computed from the linear system alone. So we only compute the solution of the linear system modulo M_2. It is still possible that some logarithms cannot be determined modulo M_2 (e.g., if 2 unknowns occur only in one equation), but usually most of the logarithms will be correctly computed modulo M_2. Then the logarithm of each factor basis element can be easily computed modulo M_1 by the Pohlig-Hellman algorithm, and the Chinese Remainder Theorem can be used to combine these with the correct modulo-M_2 logarithms to compute the logarithms modulo M of most elements of the factor basis.

Similar index-calculus-type algorithms should have techniques for handling small prime divisors of the modulus M when the solution of the linear system has high nullity modulo these small primes.

The following function `Sieve` has been developed by Allan Steel, based on code by Benjamin Costello. Its arguments are the field $K = \mathbf{F}_p$, the factor base prime limit `qlimit`, the c_1, c_2 range limit `climit`, and the desired stopping ratio of relations to unknowns. The function returns the sparse relation matrix A together with an indexed set containing the corresponding extended factor base (the small primes and the $H + c_i$ values which yield relations).

```
> function Sieve(K, qlimit, climit, ratio)
>     p := #K;
>     Z := Integers();
>     H := Iroot(p, 2) + 1;
>     J := H^2 - p;
>
>     // Get factor basis of all primes <= qlimit.
>     fb_primes := [p: p in [2 .. qlimit] | IsPrime(p)];
```

```
>
>     printf "Factor base has %o primes, climit is %o\n", #fb_primes, climit;
>
>     // Ensure that the primitive element of K is prime (as an integer).
>     a := rep{x: x in [2..qlimit] | IsPrime(x) and IsPrimitive(K!x)};
>     SetPrimitiveElement(K,K!a);
>
>     // Initialize extended factor base FB to fb_primes (starting with a).
>     FB := {@ Z!a @};
>     for x in fb_primes do
>         Include(~FB, x);
>     end for;
>
>     // Initialize A to 0 by 0 sparse matrix over Z.
>     A := SparseMatrix();
>
>     // Get logs of all factor basis primes.
>     log2 := Log(2.0);
>     logqs := [Log(q)/log2: q in fb_primes];
>
>     for c1 in [1 .. climit] do
>
>         // Stop if ratio of relations to unknowns is high enough.
>         if Nrows(A)/#FB ge ratio then break; end if;
>
>         if c1 mod 50 eq 0 then
>             printf "c1: %o, #rows: %o, #cols: %o, ratio: %o\n",
>                 c1, Nrows(A), #FB, RealField(8)!Nrows(A)/#FB;
>         end if;
>
>         // Initialize sieve.
>         sieve := [z: i in [1 .. climit]] where z := Log(1.0);
>         den := H + c1;                      // denominator of relation
>         num := -(J + c1*H);             // numerator
>
>         for i := 1 to #fb_primes do
>             // For each prime q in factor base...
>             q := fb_primes[i];
>             logq := logqs[i];
>
>             qpow := q;
>             while qpow le qlimit do
>                 // For all powers qpow of q up to qlimit...
>
>                 if den mod qpow eq 0 then break; end if;
>                 c2 := num * Modinv(den, qpow) mod qpow;
>                 if c2 eq 0 then c2 := qpow; end if;
>
```

```
>                   nextqpow := qpow*q;
>                   // Ensure c2 >= c1 to remove redundant relations.
>                   while c2 lt c1 do
>                       c2 +:= qpow;
>                   end while;
>
>                   while c2 le #sieve do
>                       // Add logq into sieve for c2.
>                       sieve[c2] +:= logq;
>
>                       // Test higher powers of q if nextqpow is too large.
>                       if nextqpow gt qlimit then
>                           prod := (J + (c1 + c2)*H + c1*c2) mod p;
>                           nextp := nextqpow;
>                           while prod mod nextp eq 0 do
>                               sieve[c2] +:= logq;
>                               nextp *:= q;
>                           end while;
>                       end if;
>                       c2 +:= qpow;
>                   end while;
>                   qpow := nextqpow;
>               end while;
>           end for;
>
>           // Check sieve for full factorizations.
>           rel := den * (H + 1);    // the relation
>           relinc := H + c1;        // add to relation to get next relation
>           count := 0;
>           for c2 in [1 .. #sieve] do
>               n := rel mod p;
>               if Abs(sieve[c2] - Ilog2(n)) lt 1 then
>                   fact, r := TrialDivision(n, qlimit);
>                   if r eq [] and (#fact eq 0 or fact[#fact][1] lt qlimit) then
>                       // Include each H + c_i in extended factor basis.
>                       Include(~FB, H + c1);
>                       Include(~FB, H + c2);
>
>                       // Include relation (H + c1)*(H + c2) = fact.
>                       row := Nrows(A) + 1;
>                       for t in fact do
>                           SetEntry(~A, row, Index(FB, t[1]), t[2]);
>                       end for;
>                       if c1 eq c2 then
>                           SetEntry(~A, row, Index(FB, H + c1), -2);
>                       else
>                           SetEntry(~A, row, Index(FB, H + c1), -1);
>                           SetEntry(~A, row, Index(FB, H + c2), -1);
```

```
>                    end if;
>                 end if;
>             end if;
>             rel +:= relinc;
>         end for;
>     end for;
>
>     // Check matrix by multiplying out relations in field.
>     assert {&*[(K!FB[j])^A[k, j]: j in Support(A, k)]: k in [1..Nrows(A)]}
>             eq {1};
>
>     return A, FB;
> end function;
```

We first apply the function to a trivial example to illustrate the main points. We let K be $\mathbf{F}_{103}$, and we make the factor basis include primes up to 35, the c_i range be up to 27, and the stopping ratio be 1.1. We first compute the relation matrix A and the extended factor basis F.

```
> K := GF(103);
> A, F := Sieve(K, 35, 27, 1.1);
Factor base has 11 primes, climit is 27
> A;
Sparse matrix with 33 rows and 30 columns over Integer Ring
```

We examine a few rows of A and the extended factor basis F. Note that 5 is the smallest prime which is primitive in K, so it has been inserted first in F.

```
> A[1]; A[2]; A[30];
(1 0 0 0 0 1 0 0 0 0 0 -1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 1 1 0 0 0 0 0 0 -1 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 0 0 0 0 0 0)
> F;
{@ 5, 2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 12, 14, 15, 16, 21, 38,
20, 26, 35, 22, 33, 34, 24, 25, 28, 30, 37, 32, 36 @}
```

Next we compute a solution vector v such that $v \cdot A^{tr} = 0$ modulo $M = \#K - 1$.

```
> Factorization(#K-1);
[ <2, 1>, <3, 1>, <17, 1> ]
> v := ModularSolution(A, #K - 1);
> v;
(1 44 39 4 61 72 70 80 24 86 57 25 48 40 74 43 22 0 1 5 3 100 12 69 2
    92 84 93 16 64)
```

Notice that 0 occurs in v, so the corresponding logarithm is not known. The vector is normalized so that the logarithm of 5 is 1, as desired. We finally compute the powers of the primitive element of K by each element of v and check that all of these match the entries of F except for the one missed entry. Note also that because M has small prime divisors, it is fortunate that the logarithms are all correct, apart from the missed one.

```
> a := PrimitiveElement(K);
```

```
> a;
5
> Z := IntegerRing();
> [a^Z!x: x in Eltseq(v)];
[ 5, 2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 12, 14, 15, 16, 21, 38, 1,
5, 35, 22, 33, 34, 24, 25, 28, 30, 37, 32, 36 ]
> F;
{@ 5, 2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 12, 14, 15, 16, 21, 38, 20,
26, 35, 22, 33, 34, 24, 25, 28, 30, 37, 32, 36 @}
```

Finally, we take $K = \mathbf{F}_p$, where p is the smallest prime above 10^{20} such that $(p-1)/2$ is prime. The sparse matrix constructed here is close to 1000×1000, but the solution vector is still found in less than a second.

```
> K := GF(100000000000000000763);
> Factorization(#K - 1);
[ <2, 1>, <50000000000000000381, 1> ]
> time A, F := Sieve(K, 2200, 800, 1.1);
Factor base has 327 primes, climit is 800
c1: 50, #rows: 222, #cols: 555, ratio: 0.4
c1: 100, #rows: 444, #cols: 738, ratio: 0.60162602
c1: 150, #rows: 595, #cols: 836, ratio: 0.71172249
c1: 200, #rows: 765, #cols: 921, ratio: 0.83061889
c1: 250, #rows: 908, #cols: 973, ratio: 0.9331963
c1: 300, #rows: 1014, #cols: 1011, ratio: 1.003
c1: 350, #rows: 1105, #cols: 1023, ratio: 1.0802
Time: 3.990
> A;
Sparse matrix with 1141 rows and 1036 columns over Integer Ring
> time v := ModularSolution(A, #K - 1);
Time: 0.170
```

We observe that the list consisting of the primitive element powered by the computed logarithms agrees with the factor basis for at least the first 30 elements.

```
> a := PrimitiveElement(K);
> a;
2
> v[1], v[2];
1 71610399209536789314
> a^71610399209536789314;
3
> P := [a^x: x in Eltseq(v)];
> [P[i]: i in [1 .. 30]];
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ]
> [F[i]: i in [1 .. 30]];
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
```

67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113]

There are in fact 8 logarithms which could not be computed correctly.

```
> [i: i in [1..#F] | P[i] ne F[i]];
[ 671, 672, 673, 737, 738, 947, 1024, 1025 ]
```

A zero value appearing in the place of a logarithm indicates that the nullity of the system was larger than 1, even considered modulo $(\#K - 1)/2$.

```
> [v[i]: i in [1..#F] | P[i] ne F[i]];
[ 0, 22076788376647522787, 73252391663364176895, 0,
33553634905886614528, 42960107136526083388, 0, 57276316725691590267 ]
> [P[i]: i in [1..#F] | P[i] ne F[i]];
[ 1, 7480000052677, 7960000056517, 1, 5220000041437,
89380028430619715763, 1, 11360000275636 ]
```

However, we have found the correct logarithms for nearly all factor basis elements. As pointed out above, the Pollard-Rho method applied to an element of the factor basis for this field would take many hours!

27.14   Bibliography

[**COS86**]   D. Coppersmith, A. M. Odlyzko, and R. Schroeppel.   Discrete logarithms in GF(p). *Algorithmica*, 1:1–15, 1986.

[**DEJ84**]   I.S. Duff, A.M. Erisman, and J.K.Reid.   *Direct methods for sparse matrices.* Monographs on Numerical Analysis. Oxford University Press, 1984.

[**HHR93**]   George Havas, Derek F. Holt, and Sarah Rees.   Recognizing badly presented Z-modules. *Linear Algebra and its Applications*, 192:137–164, 1993.

[**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*, pages 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*, pages 109–133. Springer-Verlag, 1991.

28 VECTOR SPACES

Chapter 28

VECTOR SPACES

28.1 Introduction

In this chapter we will discuss vector spaces and their linear transformations. Let K be a field. In Magma, the standard K-vector space is taken to be the set of n-tuples over the field K, which we shall write as $K^{(n)}$.

A rectangular matrix over a field K is considered to be an element of the vector space consisting of all $m \times n$ matrices over K. This vector space will be written as $K^{(m \times n)}$. Let U and V be K-vector spaces of dimensions m and n, respectively. The set of all linear transformations with domain U and codomain V will be denoted by $\mathrm{Hom}_K(U, V)$. Once bases have been chosen for U and V, we may identify $\mathrm{Hom}_K(U, V)$ with $K^{(m \times n)}$. Thus, $K^{(m \times n)}$ is first of all a vector space and all the normal vector space operations apply. However, since it is also a set of mappings, some additional operations arising from this characterization apply.

We shall use the term *vector space* or *K-vector space* (if we wish to emphasize the coefficient field) to refer to both the space $K^{(n)}$ and the space $K^{(m \times n)}$. If we wish to differentiate between the two, we shall use the term *tuple space* when referring to $K^{(n)}$ and the term *matrix space* referring to $K^{(m \times n)}$.

28.1.1 Vector Space Categories

The family of all finite dimensional vector spaces over a given field K forms a category, while the set of all finite dimensional vector spaces forms a family of categories indexed by the field K. In this family of categories, objects are vector spaces and the morphisms are linear transformations. The (indexed family of) categories consisting of vector spaces of n-tuples has the name `ModTupFld`, while the (indexed family of) categories consisting of vector spaces of $m \times n$-matrices has the name `ModMatFld`.

28.1.2 The Construction of a Vector Space

Every vector space V defined over a field K is created either as a subspace of the row space $K^{(n)}$ (tuple spaces) or as a subspace of $K^{(m \times n)}$ (matrix modules). Thus, the construction of a general vector space is a two step process:

(i) The appropriate row space $K^{(n)}$, is constructed;

(ii) The required vector space V is then defined as a subspace or quotient space of $K^{(n)}$.

28.2 Creation of Vector Spaces and Arithmetic with Vectors

28.2.1 Construction of a Vector Space

VectorSpace(K, n)

KSpace(K, n)

Given a field K and a non-negative integer n, create the n-dimensional vector space $V = K^{(n)}$, consisting of all n-tuples over K. The vector space is created with respect to the standard basis, $e_1, \ldots, e_n$, where e_i $(i = 1, \ldots, n)$ is the vector containing a 1 in the i-th position and zeros elsewhere.

Use of the functions VectorSpace and KSpace ensures that subspaces of V will be presented in embedded form.

KModule(K, n)

Given a field K and a non-negative integer n, create the n-dimensional vector space $V = K^{(n)}$, consisting of all n-tuples over K. The vector space is created with respect to the standard basis, $e_1, \ldots, e_n$, where e_i $(i = 1, \ldots, n)$ is the vector containing a 1 in the i-th position and zeros elsewhere.

Use of the function KModule ensures that subspaces of V will be presented in reduced form. In all other respects, a vector space created by this function is identical to one created by KSpace.

KMatrixSpace(K, m, n)

Given a field K and integers m and n greater than one, create the vector space $K^{(m \times n)}$, consisting of all $m \times n$ matrices over K. The vector space is created with the standard basis, $\{E_{ij} \mid i = 1 \ldots, m, j = 1 \ldots, n\}$, where E_{ij} is the matrix having a 1 in the (i, j)-th position and zeros elsewhere.

Note that for a matrix space, subspaces will always be presented in embedded form, i.e. there is no reduced mode available for matrix spaces.

Hom(V, W)

If V is the vector space $K^{(m)}$ and W is the vector space $K^{(n)}$, create the matrix space $\mathrm{Hom}_K(V, W)$ as the vector space $K^{(m \times n)}$, represented as the set of all $m \times n$ matrices over K. The vector space is created with the standard basis, $\{E_{ij} \mid i = 1 \ldots, m, j = 1 \ldots, n\}$, where E_{ij} is the matrix having a 1 in the (i, j)-th position and zeros elsewhere.

Example H28E1_____

We construct the vector space V consisting of 6-tuples over the rational field.

```
> Q := RationalField();
> V := VectorSpace(Q, 6);
> V;
Vector space of dimension 6 over Rational Field
```

Example H28E2_____

We construct the matrix space M consisting of 3×5 matrices over the field $\mathbf{Q}(\sqrt{5})$.

```
> K<w> := QuadraticField(5);
> V := KMatrixSpace(K, 3, 5);
> V;
Full Vector Space of 3 by 5 matrices over Quadratic Field Q(w)
```

28.2.2   Construction of a Vector Space with Inner Product Matrix

---

VectorSpace(K, n, F)

KSpace(K, n, F)

> Given a field K, a non-negative integer n and a square $n \times n$ symmetric matrix F, create the n-dimensional vector space $V = K^{(n)}$ (in embedded form), with inner product matrix F. This is the same as VectorSpace(K, n), except that the functions Norm and InnerProduct (see below) will be with respect to the inner product matrix F.

28.2.3   Construction of a Vector

---

elt< V | L >

> (1) Suppose V is a subspace of the vector space $K^{(n)}$. Given elements $a_1, \ldots, a_n$ belonging to K, construct the vector $v = (a_1, \ldots, a_n)$ as a vector of V. Note that if v is not an element of V, an error will result.
>
> (2) Suppose V is a subspace of the matrix space $K^{(m \times n)}$. Given elements $a_1, \ldots, a_{mn}$ belonging to K, construct the matrix $m = (a_1, \ldots, a_{mn})$ as an element of V. Note that if m is not an element of V, an error will result.

```
V ! Q
```

(1) Suppose V is a subspace of the vector space $K^{(n)}$. Given elements $a_1, \ldots, a_n$ belonging to K, construct the vector $v = (a_1, \ldots, a_n)$ as a vector of V. Note that if v is not an element of V, an error will result.

(2) Suppose V is a subspace of the matrix space $K^{(m \times n)}$. Given elements $a_1, \ldots, a_{mn}$ belonging to K, construct the matrix $m = (a_1, \ldots, a_{mn})$ as an element of V. Note that if m is not an element of V, an error will result.

```
CharacteristicVector(V, S)
```

Given a subspace V of the vector space $K^{(n)}$ together with a set S of integers lying in the interval $[1, n]$, return the characteristic number of S as a vector of V.

```
V ! 0
```
```
Zero(V)
```

The zero element for the vector space V.

```
Random(V)
```

Given a vector space V defined over a finite field, return a random vector.

Example H28E3

We create the 5-dimensional vector space V over $\mathbf{F}_4$ and define the vector $u = (1, w, 1 + w, 0, 0)$, where w is a primitive element of $\mathbf{F}_4$.

```
> K<w> := GaloisField(4);
> V     := VectorSpace(K, 5);
> u     := V ! [1, w, 1+w, 0, 0];
> u;
(1 w w + 1 0 0)
> zero := V ! 0;
> zero;
(0 0 0 0 0)
r := Random(V);
(1 0 w 1 w + 1)
```

Example H28E4

We create an element belonging to the space of 3×4 matrices over the number field $\mathbf{Q}(w)$, where w is a root of $x^7 - 7x + 3$.

```
> R<x> := PolynomialRing(RationalField());
> L<w> := NumberField(x^7 - 7*x + 3);
> L34 := KMatrixSpace(L, 3, 4);
> a := L34 ! [ 1, w, 0, -w, 0, 1+w, 2, -w^3, w-w^3, 2*w, 1/3, 1 ];
> a;
[1     w     0     -1 * w]
```

```
[0      w + 1     2      -1 * w^3]
[-1 * w^3 + w     2 * w     1/3     1]
```

28.2.4   Deconstruction of a Vector

> ```
> ElementToSequence(u)
> ```
> ```
> Eltseq(u)
> ```

Given an element u belonging to the K-vector space V, return u in the form of a sequence Q of elements of V. Thus, if u is an element of $K^{(n)}$, then $Q[i] = u[i]$, $1 \le i \le n$.

28.2.5   Arithmetic with Vectors

For the following operations the vectors u and v must belong to the same vector space i.e. the same tuple space $K^{(n)}$ or the same matrix space $K^{(m \times n)}$. The scalar a must belong to the field K.

> ```
> u + v
> ```

Sum of the vectors u and v, where u and v lie in the same vector space.

> ```
> -u
> ```

Additive inverse of the vector u.

> ```
> u - v
> ```

Difference of the vectors u and v, where u and v lie in the same vector space.

> ```
> x * u
> ```
> ```
> u * x
> ```

The scalar product of the vector u belonging to the K-vector space and the field element x belonging to K.

> ```
> u / x
> ```

The scalar product of the vector u belonging to the K-vector space and the field element $1/x$ belonging to K where x is non-zero.

> ```
> NumberOfColumns(u)
> ```
> ```
> Ncols(u)
> ```

The number of columns in the vector u.

> ```
> Depth(u)
> ```

The index of the first non-zero entry of the vector u (0 if none such).

(u, v)

InnerProduct(u, v)

Return the inner product of the vectors u and v with respect to the inner product defined on the space. If an inner product matrix F is given when the space is created, then this is defined to be $u \cdot F \cdot v^{tr}$. Otherwise, this is simply $u \cdot v^{tr}$.

IsZero(u)

Returns `true` iff the vector u belonging to a vector space is the zero element.

Norm(u)

Return the norm product of the vector u with respect to the inner product defined on the space. If an inner product matrix F is given when the space is created, then this is defined to be $u \cdot F \cdot u^{tr}$. Otherwise, this is simply $u \cdot u^{tr}$.

Normalise(u)

Normalize(u)

Given an element u, not the zero element, belonging to the K-vector space V, return $\frac{1}{a} * u$, where a is the first non-zero component of u. If u is the zero vector, it is returned. The net effect is that `Normalize(u)` always returns a vector v in the subspace generated by u, such that the first non-zero component of v (if existent) is $K!1$.

Rotate(u, k)

Given a vector u, return the vector obtained from u by rotating by k coordinate positions.

Rotate($\sim$u, k)

Given a vector u, destructively rotate u by k coordinate positions.

NumberOfRows(u)

Nrows(u)

The number of rows in the vector u (1 of course; included for completeness).

Support(u)

A set of integers giving the positions of the non-zero components of the vector u.

TensorProduct(u, v)

The tensor (Kronecker) product of the vectors u and v. The resulting vector has degree equal to the product of the degrees of u and v.

Trace(u, F)

Trace(u)

Given a vector belonging to the space $K^{(n)}$, and a subfield F of K, return the vector obtained by replacing each component of u by its trace over the subfield F. If F is the prime field of K, it may be omitted.

Weight(u)

The number of non-zero components of the vector u.

Example H28E5

We illustrate the use of the arithmetic operators for module elements by applying them to elements of the 4-dimensional vector space over the field $\mathbf{Q}(w)$, where w is an 8-th root of unity.

```
> K<w> := CyclotomicField(8);
> V := VectorSpace(K, 4);
> x := V ! [ w, w^ 2, w^ 4, 0];
> y := V ! [1, w, w^ 2, w^ 4];
> x + y;
((1 + w) (w + w^2) (-1 + w^2) -1)
> -x;
((-w) (-w^2)          1          0)
> x - y;
((-1 + w) (-w + w^2) (-1 - w^2) 1)
> w * x;
(  (w^2) (w^3)  (-w)          0)
> y * w^ -4;
(       -1 (-w) (-w^2)          1)
> Normalize(x);
(        1    (w) (w^3)          0)
> InnerProduct(x, y);
(w - w^2 + w^3)
> z := V ! [1, 0, w, 0 ];
> z;
(        1        0 (w)          0)
> Support(z);
{ 1, 3 }
```

Example H28E6

We illustrate how one can define a non-trivial inner product on a space.

```
> Q := RationalField();
> F := SymmetricMatrix(Q, [1, 0,2, 0,0,3, 1,2,3,4]);
> F;
[1 0 0 1]
[0 2 0 2]
```

```
[0 0 3 3]
[1 2 3 4]
> V := VectorSpace(Q, 4, F);
> V;
Full Vector space of degree 4 over Rational Field
Inner Product Matrix:
[1 0 0 1]
[0 2 0 2]
[0 0 3 3]
[1 2 3 4]
> v := V![1,0,0,0];
> Norm(v);
1
> w := V![0,1,0,0];
> Norm(w);
2
> InnerProduct(v, w);
0
> z := V![0,0,0,1];
> Norm(z);
4
> InnerProduct(v, z);
1
> InnerProduct(w, z);
2
```

28.2.6   Indexing Vectors and Matrices

The indexing operations have a different meaning depending upon whether they are applied to a tuple space or a matrix space.

> u[i]

> u[i, j]

> Given a vector u belonging to a K-vector space V, the result of this operation depends upon whether V is a tuple or matrix space.
>
> If V is a subspace of $K^{(n)}$, and i, $1 \leq i \leq n$, is a positive integer, the i-th component of the vector u is returned (as an element of the field K).
>
> If V is a subspace of $K^{(m \times n)}$, and i, $1 \leq i \leq m$, is a positive integer, $u[i]$ will return the i-th row of the matrix u (as an element of the vector space $K^{(n)}$). Similarly, if i and j, $1 \leq i \leq m$, $1 \leq j \leq n$, are positive integers, $u[i,j]$ will return the (i,j)-th component of the matrix u (as an element of K).

```
 u[i] := x
```
```
 u[i] := x
```
```
 u[i, j] := x
```

Given a vector u belonging to a K-vector space V, and an element x of K, the result of this operation depends upon whether V is a tuple or matrix space.

If V is a subspace of $K^{(n)}$, and i, $1 \leq i \leq n$, is a positive integer, the i-th component of the vector u is redefined to be x.

If V is a subspace of $K^{(m \times n)}$ and $1 \leq i \leq m$ is a positive integer and x is an element of $K^{(n)}$, u[i] := x will redefine the i-th row of the matrix u to be the vector x, where x must be an element of $K^{(n)}$. Similarly, if $1 \leq i \leq m$, $1 \leq j \leq n$, are positive integers, u[i, j] := x will redefine the (i, j)-th component of the matrix u to be x, where x must be an element of K.

Example H28E7_____

We illustrate the use of the indexing operators for vector space elements by applying them to a 3-dimensional tuple space and a 2×3 matrix space over the field $\mathbf{Q}(w)$, where w is an 8-th root of unity.

```
> K<w> := CyclotomicField(8);
> V := VectorSpace(K, 3);
> u := V ! [ 1 + w, w^ 2, w^ 4];
> u;
((1 + w)      (w^2)               -1)
> u[3];
-1
> u[3] := 1 + w - w^7;
> u;
((1 + w) (w^2) (1 + w + w^3))
> // We now demonstrate indexing a matrix space
> W := KMatrixSpace(K, 2, 3);
> l := W ! [ 1 - w, 1 + w, 1 + w + w^2, 0, 1 - w^7, 1 - w^3 + w^6 ];
> l;
[(1 - w) (1 + w) (1 + w + w^2)]
[0 (1 + w^3) (1 - w^2 - w^3)]
> l[2];
(0 (1 + w^3) (1 - w^2 - w^3))
> l[2,2];
(1 + w^3)
> m := l[2];
> m;
(0 (1 + w^3) (1 - w^2 - w^3))
> l[2] := u;
> l;
[(1 - w) (1 + w) (1 + w + w^2)]
[(1 + w) (w^2) -1]
> l[2, 3] := 1 + w - w^7;
```

```
> l;
[(1 - w) (1 + w) (1 + w + w^2)]
[(1 + w) (w^2) (1 + w + w^3)]
```

28.3   Subspaces, Quotient Spaces and Homomorphisms

28.3.1   Construction of Subspaces

The conventions defining the presentations of subspaces and quotient spaces are as follows:

If V has been created using the function `VectorSpace` or `MatrixSpace`, then every subspace and quotient space of V is given in terms of a basis consisting of elements of V, i.e. by means of an embedded basis.

If V has been created using the function `RModule`, then every subspace and quotient space of V is given in terms of a reduced basis.

---

sub< V | L >

Given a K-vector space V, construct the subspace U generated by the elements of V specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of K defining an element of V;

(b) A set or sequence whose terms are elements of V;

(c) A subspace of V;

(d) A set or sequence whose terms are subspaces of V.

The generators stored for U consist of the vectors specified by terms L_i together with the stored generators for subspaces specified by terms of L_i. Repetitions of a vector and occurrences of the zero vector are removed (unless U is the trivial subspace).

The constructor returns the subspace U and the inclusion homomorphism $f : U \to V$. If V is of embedded type, the basis constructed for U consists of elements of V. If V is of standard type, a standard basis is constructed for U.

Morphism(U, V)

Assuming the vector space U has been created as a subspace of V, the function returns the matrix defining the embedding of U into V.

Example H28E8

The ternary Golay code is a six-dimensional subspace of the vector space $K^{(11)}$, where K is $\mathbf{F}_3$. This subspace is first constructed in the space constructed by the `VectorSpace` function.

```
> K11 := VectorSpace(FiniteField(3), 11);
> G3 := sub< K11  |
>     [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>     [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>     [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> G3;
Vector space of degree 11, dimension 6 over GF(3)
Generators:
(1 0 0 0 0 0 1 1 1 1 1)
(0 1 0 0 0 0 0 1 2 2 1)
(0 0 1 0 0 0 1 0 1 2 2)
(0 0 0 1 0 0 2 1 0 1 2)
(0 0 0 0 1 0 2 2 1 0 1)
(0 0 0 0 0 1 1 2 2 1 0)
Echelonized basis:
(1 0 0 0 0 0 1 1 1 1 1)
(0 1 0 0 0 0 0 1 2 2 1)
(0 0 1 0 0 0 1 0 1 2 2)
(0 0 0 1 0 0 2 1 0 1 2)
(0 0 0 0 1 0 2 2 1 0 1)
(0 0 0 0 0 1 1 2 2 1 0)
```

Example H28E9

We now construct the ternary Golay code starting with the vector space constructed using the `RModule` function. In this case the subspace is presented on a reduced basis.

```
> K11 := RModule(FiniteField(3), 11);
> G3 := sub< K11 |
>     [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>     [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>     [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> G3;
KModule G3 of dimension 6 with base ring GF(3)
> Basis(G3);
[
    G3: (1 0 0 0 0 0),
    G3: (0 1 0 0 0 0),
    G3: (0 0 1 0 0 0),
    G3: (0 0 0 1 0 0),
    G3: (0 0 0 0 1 0),
    G3: (0 0 0 0 0 1)
]
> f := Morphism(G3, K11);
```

```
> f;
[1 0 0 0 0 0 1 1 1 1 1]
[0 1 0 0 0 0 0 1 2 2 1]
[0 0 1 0 0 0 1 0 1 2 2]
[0 0 0 1 0 0 2 1 0 1 2]
[0 0 0 0 1 0 2 2 1 0 1]
[0 0 0 0 0 1 1 2 2 1 0]
```

28.3.2 Construction of Quotient Vector Spaces

quo< V | L >

Given a K-vector space V, construct the quotient vector space $W = V/U$, where U is the subspace generated by the elements of V specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of K defining an element of V;

(b) A set or sequence whose terms are elements of V;

(c) A subspace of V;

(d) A set or sequence whose terms are subspaces of V.

The generators constructed for U consist of the elements specified by terms L_i together with the stored generators for subspaces specified by terms of L_i.

The constructor returns the quotient space W and the natural homomorphism $f : V \to W$.

V / U

Given a subspace U of the vector space V, construct the quotient space W of V by U. If r is defined to be $\dim(V) - \dim(U)$, then W is created as an r-dimensional vector space relative to the standard basis.

The constructor returns the quotient space W and the natural homomorphism $f : V \to W$.

Example H28E10

We construct the quotient of $K^{(11)}$ by the Golay code.

```
> K11 := VectorSpace(FiniteField(3), 11);
> Q3, f := quo< K11 |
>   [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>   [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>   [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> Q3;
Full Vector space of degree 5 over GF(3)
> f;
Mapping from: ModTupFld: K11 to ModTupFld: Q3
```

Example H28E11

If we wished to construct this quotient of $K^{(11)}$ as a subspace of the original space, we could do so using the `Complement` function.

```
> K11 := VectorSpace(FiniteField(3), 11);
> S := sub< K11 |
>     [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>     [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>     [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> Complement(K11, S);
Vector space of degree 11, dimension 5 over GF(3)
Echelonized basis:
(0 0 0 0 0 0 1 0 0 0 0)
(0 0 0 0 0 0 0 1 0 0 0)
(0 0 0 0 0 0 0 0 1 0 0)
(0 0 0 0 0 0 0 0 0 1 0)
(0 0 0 0 0 0 0 0 0 0 1)
```

Example H28E12

We construct a subspace and its quotient space in $\mathbf{Q}^{(3\times4)}$.

```
> Q := RationalField();
> Q3 := VectorSpace(Q, 3);
> Q4 := VectorSpace(Q, 4);
> H34 := Hom(Q3, Q4);
> a := H34 ! [ 2, 0, 1, -1/2,  1, 0, 3/2, 4,  4/5, 6/7, 0, -1/3];
> b := H34 ! [ 1/2, -3, 0, 5,  1/3, 2, 4/5, 0,  5, -1, 5, 7];
> c := H34 ! [ -1, 4/9, 1, -4,  5, -5/6, -3/2, 0,  4/3, 7, 0, 7/9];
> d := H34 ! [ -3, 5, 1/3, -1/2,  2/3, 4, -2, 0,  0, 4, -1, 0];
> a, b, c, d;
[   2    0    1 -1/2]
[   1    0  3/2    4]
[ 4/5  6/7    0 -1/3]

[1/2  -3    0    5]
[1/3   2  4/5    0]
[  5  -1    5    7]

[  -1  4/9    1   -4]
[   5 -5/6 -3/2    0]
[ 4/3    7    0  7/9]

[  -3    5  1/3 -1/2]
[ 2/3    4   -2    0]
[   0    4   -1    0]
> U := sub< H34 | a, b, c, d >;
> U:Maximal;
```

```
KMatrixSpace of 3 by 4 GHom matrices and dimension 4 over Rational Field
Echelonized basis:

[1     0     0     0]
[-33872/30351    -5164/10117    42559/50585    11560/10117]
[-10514/10117    -121582/70819    -8476/10117    -48292/30351]

[          0             1             0             0]
[ -7797/10117    4803/10117 12861/101170    5940/10117]
[ -7818/10117 -38214/70819  -7821/10117 -10967/10117]

[          0             0             1             0]
[ 31261/10117   28101/20234   -2157/20234   18552/10117]
[161802/50585 291399/70819   20088/10117   33419/10117]

[          0             0             0             1]
[-8624/30351  7445/10117  7696/50585  2408/10117]
[32388/50585 -3562/10117  6272/10117 27580/30351]
> W := H34/U;
> W;
Full Vector space of degree 8 over Rational Field
```

28.4 Changing the Coefficient Field

The standard constructions described in section 31.5 for R-modules may be applied to vector spaces. In addition, we may extend or restrict the field of scalars, using the functions described here.

ExtendField(V, L)

> Given a K-vector space V, with K a field and L an extension of K, construct the L-vector space $U = V \otimes_K L$. The function returns
>
> (a) the vector space U; and
>
> (b) the inclusion homomorphism $\phi : V \to U$.

RestrictField(V, L)

> Given a K-vector space V, with K a field and L a subfield of K, construct the L-vector space U consisting of those vectors of V having all of their components lying in the subfield L. The function returns
>
> (a) the vector space U; and
>
> (b) the restriction homomorphism $\phi : V \to U$.

```
VectorSpace(V, F)
```
```
KSpace(V, F)
```
```
KMatrixSpace(V, F)
```
```
KModule(V, F)
```

Given an n-dimensional K-vector space V, and a subfield F of a finite field or cyclotomic field K such that K has degree m over F, construct a vector space U of dimension mn over the field F. The function returns

(a) the vector space U; and

(b) a mapping $\phi : V \rightarrow U$ such that a vector $(v_1, \ldots, v_i, \ldots, v_n)$ of V is mapped into the vector

$$(u_{11}, \ldots, u_{1n}, \ldots, u_{i1}, \ldots, u_{in}, \ldots, u_{n1}, \ldots u_{nn}),$$

where $(u_{i1}, \ldots, u_{in})$ is the field element v_i written as a vector over the subfield F.

28.5 Basic Operations

28.5.1 Accessing Vector Space Invariants

```
V . i
```

Given a vector space V and a positive integer i, return the i-th generating element of V.

```
CoefficientField(V)
```
```
BaseField(V)
```

Given a K-vector space V, return the field K.

```
Degree(V)
```

Given a K-vector space V which is a subspace of $K^{(n)}$, return n.

```
Degree(u)
```

Given a vector u belonging to a subspace of the vector space $K^{(n)}$, return n.

```
Dimension(V)
```

The dimension of the vector space V.

```
Generators(V)
```

The generators for the vector space V, returned as a set.

NumberOfGenerators(M)

Ngens(M)

> The number of generators for the vector space V.

OverDimension(V)

> Given a K-vector space V which is a subspace of $K^{(n)}$, return n.

OverDimension(u)

> Given a vector u belonging to a subspace of the vector space $K^{(n)}$, return n.

Generic(V)

> The generic vector space containing V, i.e. the full vector space in which V is naturally embedded.

Parent(V)

> The power structure for the vector space V (the set consisting of all finite dimensional vector spaces).

28.5.2 Membership and Equality

---

v in V

> Returns true if the element v lies in the vector space V, where v and V belong to a common space.

---

v notin V

> Returns true if the element v does not lie in the vector space V, where v and V belong to a common space.

U subset V

> Returns true if the K-vector space U is contained in the K-vector space V, where U and V are subspaces of some common vector space.

U notsubset V

> Returns true if the K-vector space U is not contained in the K-vector space V, where U and V are subspaces of some common vector space.

U eq V

> Returns true if the subspaces U and V are equal, where U and V belong to a common vector space.

U ne V

> Returns true if the subspaces U and V are not equal, where U and V belong to a common vector space.

28.5.3   Operations on Subspaces

| U + V |

    Sum of the subspaces U and V, where U and V must be subspaces of a common vector space.

| U meet V |

    Intersection of the subspaces U and V, where U and V must be subspaces of a common vector space.

| U meet:= V |

    Replace U with the intersection of the subspaces U and V, where U and V must be subspaces of a common vector space.

| &meet S |

    Intersection of the subspaces of the set or sequence S, which must be subspaces of a common vector space.

| TensorProduct(U, V) |

    The tensor (Kronecker) product of the vector spaces U and V, generated by all the tensor products of elements of U by elements of V. The resulting vector space has degree equal to the product of the degrees of U and V.

| Complement(V, U) |

    Given a subspace U of the vector space V, construct a complement for U in V (a subspace of V).

    Note that this complement is always with respect to the standard inner product, and thus need not be orthogonal with respect to a given one.

| Transversal(V, U) |

    Given a subspace U of the vector space V over a finite field, return a transversal for U in V as a set of vectors.

28.6 Reducing Vectors Relative to a Subspace

ReduceVector(W, v)

>   (Function.) Given a vector v from a tuple module V and a submodule W of V, return the reduction of v with respect to W (that is, the canonical representative of the coset $v + W$).
>
>   Note that the reduction is done with respect to the standard inner product.

ReduceVector(W, $\sim$v)

>   (Procedure.) Given a vector v from a tuple module V and a submodule W of V, replace v with its reduction of with respect to W (that is, the canonical representative of the coset $v + W$).
>
>   Note that the reduction is done with respect to the standard inner product.

DecomposeVector(U, v)

>   Given a vector v from a tuple module V and a submodule U of V, return the unique u in U and w in the complement to U in $U+ <v>$ such that $v = u + w$.
>
>   Note that the reduction is done with respect to the standard inner product, and the complement noted above need not be orthogonal.

28.7 Bases

This section is concerned with the construction of bases for vector spaces.

VectorSpaceWithBasis(Q)

VectorSpaceWithBasis(a)

KSpaceWithBasis(Q)

KSpaceWithBasis(a)

KModuleWithBasis(Q)

>   Create a vector space having as basis the terms of B (rows of a).

Basis(V)

>   The current basis for the vector space V, returned as a sequence of vectors.

BasisElement(V, i)

>   The i-th basis element for the vector space V.

BasisMatrix(V)

>   The current basis for the vector space V, returned as the rows of a matrix belonging to the matrix space $K^{(m \times n)}$, where m is the dimension of V and n is the over-dimension of V.

Coordinates(V, v)

Given a vector v belonging to the r-dimensional K-vector space V, with basis $v_1, \ldots, v_r$, return a sequence $[a_1, \ldots, a_r]$ of elements of K giving the coordinates of v relative to the V-basis: $v = a_1 * v_1 + \cdots + a_r * v_r$.

Dimension(V)

The dimension of the vector space V.

ExtendBasis(Q, U)

Given a sequence Q containing r linearly independent vectors belonging to the vector space U, extend the vectors of Q to a basis for U. The basis is returned in the form of a sequence T such that $T[i] = Q[i], i = 1, \ldots r$.

ExtendBasis(U, V)

Given an r-dimensional subspace U of the vector space V, return a basis for V in the form of a sequence T of elements such that the first r elements correspond to the given basis vectors for U.

IsIndependent(S)

Given a set S of elements belonging to the vector space V, return **true** if the elements of S are linearly independent.

IsIndependent(Q)

Given a sequence Q of elements belonging to the vector space V, return **true** if the terms of Q are linearly independent.

Example H28E13_____

These operations will be illustrated in the context of the subspace $G3$ of the 11-dimensional vector space over $\mathbf{F}_3$ defining the ternary Golay code.

```
> V11 := VectorSpace(FiniteField(3), 11);
> G3  := sub< V11 |  [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>                    [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>                    [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> Dimension(G3);
    6
> Basis(G3);
[
    (1 0 0 0 0 0 1 1 1 1 1),
    (0 1 0 0 0 0 0 1 2 2 1),
    (0 0 1 0 0 0 1 0 1 2 2),
    (0 0 0 1 0 0 2 1 0 1 2),
    (0 0 0 0 1 0 2 2 1 0 1),
    (0 0 0 0 0 1 1 2 2 1 0)
]
> S := ExtendBasis(G3, V11);
```

```
> S;
[
    (1 0 0 0 0 0 1 1 1 1 1),
    (0 1 0 0 0 0 0 1 2 2 1),
    (0 0 1 0 0 0 1 0 1 2 2),
    (0 0 0 1 0 0 2 1 0 1 2),
    (0 0 0 0 1 0 2 2 1 0 1),
    (0 0 0 0 0 1 1 2 2 1 0),
    (0 0 0 0 0 0 1 0 0 0 0),
    (0 0 0 0 0 0 0 1 0 0 0),
    (0 0 0 0 0 0 0 0 1 0 0),
    (0 0 0 0 0 0 0 0 0 1 0),
    (0 0 0 0 0 0 0 0 0 0 1)
]
> C3:= Complement(V11, G3);
> C3;
Vector space of degree 11, dimension 5 over GF(3)
Echelonized basis:
(0 0 0 0 0 0 1 0 0 0 0)
(0 0 0 0 0 0 0 1 0 0 0)
(0 0 0 0 0 0 0 0 1 0 0)
(0 0 0 0 0 0 0 0 0 1 0)
(0 0 0 0 0 0 0 0 0 0 1)
> G3 + C3;
Full Vector space of degree 11 over GF(3)
> G3 meet C3;
Vector space of degree 11, dimension 0 over GF(3)
> x := Random(G3);
> x;
(1 1 2 0 0 1 1 1 1 2 0)
> c := Coordinates(G3, x);
> c;
[ 1, 1, 2, 0, 0, 1 ]
> G3 ! &+[ c[i] * G3.i : i in [1 .. Dimension(G3)]];
(1 1 2 0 0 1 1 1 1 2 0)
```

28.8   Operations with Linear Transformations

Throughout this section, V is a subspace of $K^{(m)}$, W is a subspace of $K^{(n)}$ and a is a linear transformation belonging to $\text{Hom}_K(V, W)$. See also the chapter on general matrices for many other functions applicable to such matrices (e.g., `EchelonForm`).

`v * a`

`a(v)`

> Given an element v belonging to the vector space V, and an element a belonging to $\text{Hom}_K(V, W)$, return the image of v under the linear transformation a as an element of the vector space W.

`a * b`

> Given a matrix a belonging to $K^{(m \times n)}$ and a matrix b belonging to $K^{(n \times p)}$, for some integers m, n, p, form the product of a and b as an element of $K^{(m \times p)}$.

`Domain(a)`

> The domain of the linear transformation a belonging to $\text{Hom}_K(V, W)$, returned as a subspace of V.

`Codomain(a)`

> The codomain of the linear transformation a belonging to $\text{Hom}_K(V, W)$, returned as a subspace of W.

`Image(a)`

> The image of the linear transformation a belonging to $\text{Hom}_K(V, W)$, returned as a subspace of W.

`Rank(a)`

> The dimension of the image of the linear transformation a, i.e., the rank of the matrix a.

`Kernel(a)`

`NullSpace(a)`

> The kernel of the linear transformation a belonging to $\text{Hom}_K(V, W)$, returned as a subspace of V.

`Cokernel(a)`

> The cokernel of the linear transformation a belonging to $\text{Hom}_K(V, W)$.

Example H28E14_____

We illustrate the map operations for matrix spaces in the following example:

```
> Q := RationalField();
> Q2 := VectorSpace(Q, 2);
> Q3 := VectorSpace(Q, 3);
> Q4 := VectorSpace(Q, 4);
> H23 := Hom(Q2, Q3);
> H34 := Hom(Q3, Q4);
> x := Q2 ! [ -1, 2 ];
> a := H23 ! [ 1/2, 3, 0,  2/3, 4/5, -1 ];
> a;
[1/2   3   0]
[2/3 4/5  -1]
> Domain(a);
Full Vector space of degree 2 over Rational Field
> Codomain(a);
Full Vector space of degree 3 over Rational Field
> x*a;
( 5/6 -7/5   -2)
> b := H34 ! [ 2, 0, 1, -1/2,  1, 0, 3/2, 4,  4/5, 6/7, 0, -9/7];
> b;
[   2    0    1 -1/2]
[   1    0  3/2    4]
[ 4/5  6/7    0 -9/7]
> c := a*b;
> c;
[     4        0        5     47/4]
[   4/3     -6/7    28/15  436/105]
> x*c;
(     -4/3      -12/7     -19/15  -1447/420)
> Image(c);
Vector space of degree 4, dimension 2 over Rational Field
Echelonized basis:
(     1        0      5/4   47/16)
(     0        1    -7/30  -11/40)
> Kernel(c);
Vector space of degree 2, dimension 0 over Rational Field
> Rank(c);
2
> EchelonForm(c);
[     1        0      5/4   47/16]
[     0        1    -7/30  -11/40]
```

29 POLAR SPACES

Chapter 29
POLAR SPACES

29.1 Introduction

This chapter describes MAGMA functions for working with quadratic, bilinear and sesquilinear forms defined on vector spaces. The emphasis is on vector spaces defined over finite fields but in some instances the functions apply more widely. For quadratic forms defined on lattices see Chapter 33. For the interpretation of reflexive forms as algebras with involution see Chapter 92. General references for this material are [Bou07, Tay92].

29.2 Reflexive Forms

Let V be a vector space of dimension n over a field K. If σ is an automorphism of K, a σ-sesquilinear form on the vector space V over K is a map $\beta : V \times V \to K$ such that

$$\beta(u_1 + u_2, v) = \beta(u_1, v) + \beta(u_2, v),$$
$$\beta(u, v_1 + v_2) = \beta(u, v_1) + \beta(u, v_2)$$

and

$$\beta(au, bv) = a\sigma(b)\beta(u, v).$$

for all $u, u_1, u_2, v, v_1, v_2 \in V$ and all $a, b \in K$. If σ is the identity, the form is said to be *bilinear*.

A linear transformation g of V is an *isometry* if g preserves β; it is a *similarity* if it preserves β up to a non-zero scalar multiple.

A σ-sesquilinear form β is *reflexive* if for all $u, v \in V$, $\beta(u, v) = 0$ implies $\beta(v, u) = 0$. Any non-zero multiple of a reflexive form is again reflexive with the same group of isometries. By a theorem of Brauer [Bra36] (but sometimes referred to as the Birkhoff–von Neumann theorem), up to a non-zero scalar multiple, there are three types of *non-degenerate* reflexive forms:

Alternating. In this case σ is the identity, $\beta(u, u) = 0$ for all $u \in V$ and consequently $\beta(u, v) = -\beta(v, u)$ for all $u, v \in V$. The group of isometries is a *symplectic* group.

Symmetric. In this case σ is the identity and $\beta(u, v) = \beta(v, u)$ for all $u, v \in V$. If the characteristic of K is not two, the group of isometries is an *orthogonal* group. If the characteristic is two, the form is either alternating or pseudo-alternating (see below).

Hermitian. In this case σ is an automorphism of order two and $\beta(u, v) = \sigma\beta(v, u)$ for all $u, v \in V$. The group of isometries is a *unitary* group.

If V is a vector space V and if β is a reflexive form defined on V, the partially ordered set of totally isotropic subspaces with respect to β is often referred to as a *polar space*.

Similarly, there are polar spaces associated with quadratic forms (see Section 29.2.1). But throughout this chapter by polar space we shall simply mean a vector space furnished with either a reflexive σ-sesquilinear form or a quadratic form. See [Bue95, Chap. 2] for an account of polar spaces in a more general context.

Let K_0 be the fixed field of σ. Multiplying an alternating, symmetric or hermitian form by a non-zero element of K_0 leaves the type of the form unchanged.

However, multiplying an hermitian form by a non-zero element of K produces a sesquilinear form ξ and an element $\varepsilon \in K$ such that for all $u, v \in V$, $\xi(v, u) = \varepsilon \sigma \xi(u, v)$, where $\varepsilon \sigma(\varepsilon) = 1$. In this case ξ is said to be ε-hermitian.

Skew-hermitian. A reflexive σ-sesquilinear form is skew-hermitian if the order of σ is two and $\xi(v, u) = -\sigma \xi(v, u)$ for all $u, v \in V$. If β is hermitian and if $d \in K$ is chosen so that $d \neq \sigma(d)$, then $e = d - \sigma(d)$ satisfies $\sigma(e) = -e$. Thus $\xi(u, v) = e\beta(u, v)$ is skew-hermitian. The group of isometries of ξ coincides with the group of isometries of β and it is therefore a *unitary* group.

In the case of fields of characteristic two there is no distinction between hermitian and skew-hermitian forms and moreover, every alternating form is symmetric.

Pseudo-alternating. A symmetric form (in characteristic two) which is not alternating is said to be *pseudo-alternating*.

The three types of forms—alternating, symmetric and hermitian—correspond to the three types of classical groups of isometries: symplectic, orthogonal and unitary. But this is not quite the whole story because it does not include orthogonal groups over fields of characteristic two. In order to include these groups it is necessary to consider quadratic forms in addition to symmetric bilinear forms.

29.2.1   Quadratic Forms

If β is a bilinear form, a *quadratic form* with *polar form* β is a function $Q : V \to K$ such that

$$Q(av) = a^2 Q(v)$$

and

$$\beta(u, v) = Q(u + v) - Q(u) - Q(v)$$

for all $u, v \in V$ and all $a \in K$. We have $\beta(v, v) = 2Q(v)$ and therefore, if the characteristic of K is not two, β determines Q.

We extend the notion of polar space to include vector spaces V with an associated quadratic form Q. The pair (V, Q) is an orthogonal geometry and V is a *quadratic* space.

29.3 Inner Products

Every vector space V in MAGMA created via the `VectorSpace` intrinsic (or its synonym `KSpace`) has an associated bilinear form which is represented by a matrix and which can be accessed via `InnerProductMatrix(V)` or via the attribute `ip_form`. By default the inner product matrix is the identity. If the dimension of V is n, then any $n \times n$ matrix defined over the base field of V can serve as the inner product matrix by passing it to `VectorSpace` as an additional parameter.

If e_1, e_2, ..., e_n is a basis for V, the matrix of the form β with respect to this basis is $J := \big(\beta(e_i, e_j)\big)$.

Example H29E1

```
> K := GF(11);
> J := Matrix(K,3,3,[1,2,3, 4,5,6, 7,8,9]);
> V := VectorSpace(K,3,J);
> InnerProductMatrix(V);
[ 1  2  3]
[ 4  5  6]
[ 7  8  9]
```

A vector space may also have an associated quadratic form. This can be assigned via the function `QuadraticSpace` described in Section 29.6.3 and, if assigned, it can be accessed as the return value of `QuadraticFormMatrix`.

If e_1, e_2, ..., e_n is a basis for V, the matrix of the quadratic form q with polar form β with respect to this basis is the upper triangular matrix Q whose ith diagonal entry is $q(e_i)$ and whose (i,j)th entry for $i < j$ is $\beta(e_i, e_j)$.

In addition, in order to accommodate hermitian forms, a vector space of type `ModTupFld` has an attribute `Involution`. This attribute is intended to hold an automorphism (of order two) of the base field.

> **DotProduct(u, v)**

> If V is the generic space of the parent of u and v, let σ be the field automorphism `V‘Involution` if this attribute is assigned or the identity automorphism if `V‘Involution` is not assigned. If J is the inner product matrix of V, the expression `DotProduct(u,v)` evaluates to $uJ\sigma(v^{\mathrm{tr}})$. That is, it returns $\beta(u,v)$, where β is a bilinear or sesquilinear form on V.

> **DotProductMatrix(W)**

> The matrix of inner products of the vectors in the sequence W. The inner products are calculated using `DotProduct` and therefore take into account any field automorphism attached to the `Involution` attribute of the generic space of the universe of S.

GramMatric(V)

If B is the basis matrix of V and if J is the inner product matrix, this function returns BJB^{tr}. In this case the `Involution` attribute is ignored.

InnerProductMatrix(V)

The inner product matrix attached to the generic space of V. This is the attribute `V'ip_form`.

Example H29E2

This example illustrates the difference between `GramMatrix` and `InnerProductMatrix`. The function `GramMatrix` uses the echelonised basis of the subspace W. To obtain the matrix of inner products between a given list of vectors, use `DotProductMatrix`.

```
> K<a> := QuadraticField(-2);
> J := Matrix(K,3,3,[1,2,1, 2,1,0, 1,0,2]);
> V := VectorSpace(K,3,J);
> W := sub<V| [a,a,a], [1,2,3]>;
> InnerProductMatrix(W);
[1 2 1]
[2 1 0]
[1 0 2]
> GramMatrix(W);
[1 0]
[0 9]
> DotProductMatrix([W.1,W.2]);
[ -20 19*a]
[19*a   37]
```

Example H29E3

Continuing the previous example, the vector space V does not have the attribute `Involution` assigned and therefore `DotProduct` uses the *symmetric* bilinear form represented by the inner product matrix J. However, the field K has a well-defined operation of complex conjugation and so `InnerProduct` uses the *hermitian* form represented by J.

```
> u := W.1+W.2;
> DotProduct(u,u);
38*a + 17
> InnerProduct(u,u);
57
```

29.3.1   Orthogonality

If β is any bilinear or sesquilinear form, the vectors u and v are *orthogonal* if $\beta(u, v) = 0$. The *left orthogonal complement* of a subset X of V is the subspace

$$^{\perp}X := \{\, u \in V \mid \beta(u, x) = 0 \text{ for all } x \in X \,\}$$

and the *right orthogonal complement* of W is

$$X^{\perp} := \{\, u \in V \mid \beta(x, u) = 0 \text{ for all } x \in X \,\}.$$

If β is reflexive, then $^{\perp}X = X^{\perp}$.

OrthogonalComplement(V, X : *parameters***)**

  Right   BoolElt   *Default :* **false**

      The default value is the left orthogonal complement of X in V. To obtain the right orthogonal complement set **Right** to **true**.

Radical(V : *parameters***)**

  Right   BoolElt   *Default :* **false**

      The left radical of the inner product space V, namely $^{\perp}V$. To obtain the right radical set **Right** to **true**.

A bilinear or sesquilinear form β is *non-degenerate* if $\mathrm{rad}(V) = 0$, where V is the polar space of β.

IsNondegenerate(V)

      Returns **true** if the determinant of the matrix of inner products of the basis vectors of V is non-zero, otherwise **false**. This function takes into account the field automorphism, if any, attached to the **Involution** attribute of the generic space of V.

If V is a quadratic space over a perfect field of characteristic 2, the restriction of the quadratic form Q to the radical is a semilinear functional (with respect to $x \mapsto x^2$) whose kernel is the *singular radical* of V. A quadratic space is *non-singular* if its singular radical is zero.

SingularRadical(V)

      The kernel of the restriction of the quadratic form of the quadratic space V to the radical of V.

IsNonsingular(V)

      Returns **true** if V is a non-singular quadratic space, otherwise **false**.

29.4   Isotropic and Singular Vectors and Subspaces

Let β be a reflexive bilinear or a sesquilinear form on the vector space V. A non-zero vector v is *isotropic* (with respect to β) if $\beta(v,v) = 0$. If Q is a quadratic form, a non-zero vector v is *singular* if $Q(v) = 0$.

HasIsotropicVector(V)

> Determine whether the polar space V contains an isotropic vector; if it does, the second return value is a representative.

HasSingularVector(V)

> Determine whether the quadratic space V contains a singular vector; if it does, the second return value is a representative.

An ordered pair of vectors (u, v) such that u and v are isotropic and $\beta(u, v) = 1$ is a *hyperbolic pair*. If V is a quadratic space, u and v are required to be singular.

HyperbolicPair(V, u)

> Given a singular or isotropic vector u which is not in the radical, return a vector v such that (u, v) is a hyperbolic pair.

If V is the direct sum of subspaces U and W and if $\beta(u, w) = 0$ for all $u \in U$ and all $w \in W$, we write $V = U \perp W$.

A vector space V furnished with a reflexive form β has a direct sum decomposition $V = U \perp \text{rad}(V)$, where U is any complement to $\text{rad}(V)$ in V.

If V is a polar space, it has a *hyperbolic splitting*; namely, it is a direct sum

$$V = L_1 \perp L_2 \perp \cdots \perp L_m \perp W$$

where the L_i are 2-dimensional subspaces spanned by hyperbolic pairs and m is maximal. If the form defining the polar space is non-degenerate and not pseudo-alternating, then every isotropic (resp. singular) vector belongs to a hyperbolic pair and consequently W does not contain any isotropic (resp. singular) vectors. In this case the integer m is the *Witt index* of the form and W is called the *anisotropic component* of the splitting. A non-degenerate form on V is said to have *maximal Witt index* if $\dim V$ is $2m$ or $2m + 1$.

Example H29E4_____

The vector space of dimension 2 over $\mathbf{F}_2$ is pseudo-symplectic (the form is the identity matrix). It has three non-zero elements only one of which is isotropic. This confirms that not every isotropic vector in a non-degenerate pseudo-symplectic space belongs to a hyperbolic pair.

```
> V := VectorSpace(GF(2),2);
> IsPseudoSymplecticSpace(V);
true
> IsNondegenerate(V);
true
> { v : v in V | v ne V!0 and DotProduct(v,v) eq 0};
```

```
{
    (1 1)
}
```

HyperbolicSplitting(V)

A maximal list of pairwise orthogonal hyperbolic pairs together with a basis for the orthogonal complement of the subspace they span. This function requires the form to be non-degenerate and, except for symplectic spaces, the base ring of V must be a finite field.

Example H29E5_____

Find the hyperbolic splitting of a polar space defined by a symmetric bilinear form. In this example W is a non-degenerate subspace of the polar space V.

```
> K<a> := GF(7,2);
> J := Matrix(K,3,3,[1,2,1, 2,1,0, 1,0,2]);
> V := VectorSpace(K,3,J);
> W := sub<V| [a,a,a], [1,2,3]>;
> IsNondegenerate(W);
true
> HyperbolicSplitting(W);
<[
    [
        (a^20    1 a^39),
        (a^12    2    a)
    ]
], []>
```

Example H29E6_____

The polar space V of the previous example is degenerate and so `HyperbolicSplitting` cannot be applied directly. Instead, we first split off the radical.

```
> IsNondegenerate(V);
false
> R := Radical(V);
> H := (Dimension(R) eq 0) select V else
>    sub<V|[e : e in ExtendBasis(B,V) | e notin B] where B is Basis(R)>;
> HyperbolicSplitting(H);
<[
    [
        (   0 a^20    1),
        (   0 a^12    2)
    ]
], []>
```

A polar space V of dimension $2m$ and Witt index m has a *symplectic basis*; i.e., a basis $e_1, f_1, \ldots, e_m, f_m$ where the pairs of vectors (e_i, f_i) are mutually orthogonal hyperbolic pairs.

SymplecticBasis(V,U,W)

> Given totally isotropic subspaces U and W of a polar space V such that $V = U \oplus W$, return a symplectic basis for V such that $e_1, e_2, \ldots, e_m$ is a basis for U and $f_1, f_2, \ldots, f_m$ is a basis for W.

A subspace W of a polar space V is *totally isotropic* if every non-zero vector of W is isotropic. If V is a quadratic space, W is *totally singular* if every non-zero vector of W is singular.

IsTotallyIsotropic(V)

> Returns `true` if the polar space V is totally isotropic, otherwise `false`.

If Q is a quadratic form defined on a vector space V, a vector $v \in V$ is said to be *singular* if $Q(v) = 0$. A subspace W is *totally singular* if $Q(w) = 0$ for all $w \in W$. The *Witt index* of Q is the dimension of a maximal totally singular subspace. If the characteristic of the field is not 2 and if β is the polar form of Q, a subspace is totally singular if and only if it is totally isotropic with respect to β. Hence in this case the Witt index of Q coincides with the Witt index of β.

IsTotallySingular(V)

> Returns `true` if the quadratic space V is totally singular, otherwise `false`.

Suppose that $V = L_1 \perp \cdots \perp L_m \perp W \perp \mathrm{rad}(V)$ where the L_i are 2-dimensional subspaces spanned by hyperbolic pairs (e_i, f_i) for $1 \le i \le m$. The subspaces $P = \langle e_1, \ldots, e_m \rangle$ and $N = \langle f_1, \ldots, f_m \rangle$ are totally isotropic and we call the 4-tuple $(\mathrm{rad}(V), P, N, W)$ a *Witt decomposition* of V. A polar space is *hyperbolic* if it is the direct sum of two totally isotropic (resp. totally singular) subspaces; in Bourbaki [Bou07, p. 66] the corresponding form is said to be *neutral*.

WittDecomposition(V)

> The Witt decomposition of the space V.

WittIndex(V)

> The Witt index of the polar space V; namely half the dimension of a maximal hyperbolic subspace.

MaximalTotallyIsotropicSubspace(V)

> A representative maximal totally isotropic subspace of the polar space V.

MaximalTotallySingularSubspace(V)

> A representative maximal totally singular subspace of the quadratic space V.

> Given a quadratic space with quadratic form $q : V \to F$, the *metabolic space* based on V is the quadratic space $M = V \oplus V^*$ with quadratic form $Q : M \to F$ defined by $Q(v, f) = q(v) + vf$.

MetabolicSpace(V)

> The metabolic space based on the quadratic space V.

29.5   The Standard Forms

This section describes the "standard" alternating, hermitian, quadratic and symmetric forms defined on a finite dimensional vector space over a field. These are forms of maximal Witt index together with the quadratic forms of non-maximal Witt index over finite fields (see Section 29.4). Except for the orthogonal groups the standard forms are preserved by the MAGMA implementation of the classical groups over finite fields.

If J is the matrix of a bilinear form, X preserves the form if $XJX^{\mathrm{tr}} = J$. If J represents an hermitian form and if $\overline{X}$ is obtained from X by application of the field automorphism, then X preserves the form if $XJ\overline{X}^{\mathrm{tr}} = J$.

If β is a non-degenerate alternating form, then $\mathrm{rad}(V)$ and the anisotropic component of a hyperbolic splitting are zero. Thus the dimension of V must be even and V has a basis of mutually orthogonal hyperbolic pairs. In particular, up to equivalence, there is only one non-degenerate alternating form on V.

StandardAlternatingForm(n,R)

StandardAlternatingForm(n,q)

> If $n = 2m$, this function returns the $n \times n$ matrix of a non-degenerate alternating form over the ring R (or the field of q elements) such that if e_1, e_2, $\ldots$, e_{2m} is the standard basis, then (e_1, e_{2m}), (e_2, e_{2m-1}), $\ldots$, (e_m, e_{m+1}) are mutually orthogonal hyperbolic pairs.
>
> The group of isometries of this form is the symplectic group $\mathrm{Sp}(2m, R)$.

Example H29E7_____

Create a symplectic geometry with the standard alternating form and then check that every non-zero vector is isotropic.

```
> K := GF(5);
> J := StandardAlternatingForm(4,K);
> J;
[0 0 0 1]
[0 0 1 0]
[0 4 0 0]
[4 0 0 0]
```

```
> V := VectorSpace(K,4,J);
> forall{ v : v in V | DotProduct(v,v) eq 0 };
true
```

StandardPseudoAlternatingForm(n,K)

StandardPseudoAlternatingForm(n,q)

> The matrix of the standard pseudo-alternating form of degree n over the field K (or the finite field of order q), which must have characteristic 2; that is, a symmetric form which is not alternating.

StandardHermitianForm(n,K)

StandardHermitianForm(n,q)

> The first return value of this function is the $n \times n$ anti-diagonal matrix $(\delta_{i,n-i+1})$ over the field K (or the field of q^2 elements). If K is the finite field of q^2 elements, the second return value is the field involution $K \to K : x \mapsto x^q$. If K is a field which admits the operation of complex conjugation, the second return value is the field automorphism which sends each element to its complex conjugate.

> If β is a non-degenerate hermitian form over a finite field, then $\mathrm{rad}(V)$ is zero and the dimension of the anisotropic component of a hyperbolic splitting is either 1 or 0.

> In the finite field case, the group of isometries of this form is $\mathrm{GU}(n,q)$.

StandardQuadraticForm(n, K : *parameters*)

StandardQuadraticForm(n, q : *parameters*)

Minus	BOOLELT	*Default :* false
Variant	MONSTGELT	*Default :* "Default"

> An $n \times n$ upper triangular matrix representing a quadratic form over the field K (or the field of order q). The default option is to return a form of maximal Witt index, namely the upper triangular matrix whose non-zero entries are $\delta_{i,n-i+1}$, where $1 \le i \le (n+1)/2$. Use the Revised option of Variant to obtain the form preserved by MAGMA's implementation of the corresponding orthogonal group.

> If Minus is true and $n = 2m$, this function returns a form whose Witt index is $m - 1$. If Minus is true, $n = 2m + 1$ and K is a finite field of odd characteristic, the return value is a form of Witt index m which is similar to but not congruent to the default form.

> For historical reasons, over a finite field K of odd characteristic, in dimension $n = 2m + 1$, the $(m + 1, m + 1)$ element in the quadratic form preserved by the orthogonal groups GO(n,K) and OldGO(n,K) is $1/4$. This is the form returned when the Variant option is set to Revised or Original.

> If K is a finite field and W is the anisotropic component of a hyperbolic splitting of a form of even rank and Minus type, then W has basis vectors e and f such that

$Q(e) = \beta(e, f) = 1$ and $Q(f) = a$, where $x^2 + x + a$ is an irreducible polynomial in $K[x]$. This is the form returned by the `Default` option for `Variant`. If the characteristic of K is two, this matrix is also returned by the `Revised` option. However, if the characteristic of K is odd, the `Revised` option returns a form corresponding to an orthonormal basis for W. The `Original` option returns the form preserved by `OldGOMinus(2*m,q)`.

Example H29E8

Construct a standard quadratic form of minus type.

```
> K<z> := GF(7,2);
> Q := StandardQuadraticForm(4,49 : Minus);
> Q;
[   0    0    0    1]
[   0    1    1    0]
[   0    0 z^23    0]
[   0    0    0    0]
> _<x> := PolynomialRing(K);
> a := Q[3,3];
> IsIrreducible(x^2+x+a);
true
```

Example H29E9

Compare the revised form with the standard form: the forms `Q` above and `QR` below have different entries in the central 2×2 block.

```
> QR := StandardQuadraticForm(4,49 : Minus, Variant := "Revised");
> QR;
[   0    0    0    1]
[   0    4    0    0]
[   0    0 z^11    0]
[   0    0    0    0]
```

`StandardSymmetricForm(n, K)`

`StandardSymmetricForm(n, q : parameters)`

Minus	BOOLELT	*Default :* `false`
Variant	MONSTGELT	*Default :* "Default"

In all cases this is $Q + Q^{\mathrm{tr}}$, where Q is the corresponding standard quadratic form, as defined above.

29.6 Constructing Polar Spaces

If J is an $n \times n$ matrix, the command `VectorSpace(K,n,J)` creates a vector space of dimension n over K with a bilinear form whose matrix is J.

The default form attached to every vector space in MAGMA is the symmetric form whose matrix is the identity matrix.

A vector space V is recognised as a polar space if any of the following conditions apply. (There is no check to ensure that the inner product matrix is is non-degenerate.)

1. There is a quadratic form attached to V.

2. There is a field involution attached to V and the inner product matrix of V is hermitian or skew-hermitian with respect to this involution.

3. The inner product matrix of V is symmetric or alternating.

Thus a vector space with a symmetric inner product matrix but no quadratic form attached is a polar space. If the characteristic of the field is 2 and the form is not alternating it is a pseudo-symplectic space, otherwise we shall call it an orthogonal space to distinguish it from quadratic spaces.

    IsPolarSpace(V)

> Check if the vector space V is a quadratic space or if the Gram matrix of V is a reflexive form.

    PolarSpaceType(V)

> The type of the polar space V, returned as a string.

Example H29E10_____

Create the standard vector space of dimension 4 over the rational field and check if it is a polar space.

```
> V := VectorSpace(Rationals(),4);
> IsPolarSpace(V);
true
> PolarSpaceType(V);
orthogonal space
```

29.6.1 Symplectic Spaces

    SymplecticSpace(J)

> The symplectic space of dimension n defined by the $n \times n$ matrix J. This function checks to ensure that J is alternating.

IsSymplecticSpace(W)

> Returns **true** if the **Involution** attribute of the generic vector space W is not assigned and the space carries an alternating form, otherwise **false**.
>
> Note that a quadratic space over a field of characteristic 2 satisfies these conditions and consequently this function will return **true** for these spaces.

IsPseudoSymplecticSpace(W)

> Given a vector space W over a finite field, this intrinsic returns **true** if the base field has characteristic 2, the **Involution** attribute is not assigned to the generic space and the form is symmetric but not alternating, otherwise **false**.

DirectSum(V,W)

> If V and W are symplectic spaces this function returns the orthogonal direct sum $V \perp W$ and linear embeddings $V \to V \perp W$ and $W \to V \perp W$.

29.6.2 Unitary Spaces

In order to accommodate hermitian forms it is necessary to assign a field automorphism of order two to the **Involution** attribute of the vector space.

Thus a unitary space is characterised as a vector space V whose ambient space, **Generic(V)**, has the attribute **Involution** and whose inner product matrix is either hermitian or skew hermitian.

UnitarySpace(J, sigma)

> The n-dimensional unitary space over the base field K of J, where σ is an automorphism of K of order 2 and where J is an $n \times n$ matrix which is hermitian or skew-hermitian with respect to σ.

IsUnitarySpace(W)

> Return **true** if the **Involution** attribute of the generic space of W is assigned and the form is either hermitian or skew-hermitian when restricted to W. The second return value is 1 if the form is hermitian and -1 if the form is skew-hermitian.

Example H29E11_____

Create a unitary geometry with the standard hermitian form and check that the given vector is isotropic. Note that the function **DotProduct** takes both the form and the field involution into account when calculating its values. For finite fields, the function **InnerProduct** *ignores* the field involution.

```
> K<z> := GF(25);
> J, sigma := StandardHermitianForm(5,K);
> J;
[   0   0   0   0   1]
[   0   0   0   1   0]
[   0   0   1   0   0]
[   0   1   0   0   0]
```

```
[  1    0    0    0    0]
> sigma(z);
z^5
> V := UnitarySpace(J,sigma);
> u := V![1,z,0,z^2,-1];
> DotProduct(u,u);
0
> InnerProduct(u,u);
z^20
```

ConjugateTranspose(M, sigma)

> The transpose of the matrix $\sigma(M)$, where σ is an automorphism of the base field of the matrix M.

29.6.3   Quadratic Spaces

A vector space V with an attached quadratic form is called a *quadratic space*. The polar form of a quadratic space is the inner product matrix J of the space. If the characteristic of the field is not 2, the value of the quadratic form on a row vector v is $\frac{1}{2} v * J * v^{\mathrm{tr}}$.

QuadraticSpace(Q)

> The quadratic space of dimension n defined by the quadratic form represented by an upper triangular $n \times n$ matrix Q. The inner product matrix of the space is $Q + Q^{\mathrm{tr}}$. If Q is not upper triangular, the space will be constructed but there is no guarantee that other functions in this package will return correct results.

QuadraticSpace(f)

> The quadratic space of dimension n whose quadratic form is given by the quadratic polynomial f in n variables. If the variables are x_1, ..., x_n, then for $i \leq j$, the (i, j)-th entry of the matrix of the form is the coefficient of $x_i x_j$ in f.

SymmetricToQuadraticForm(J)

> Provided the characteristic of the field is not two, this is the upper triangular matrix which represents the same quadratic form as the symmetric matrix J.

QuadraticFormMatrix(V)

> The (upper triangular) matrix which represents the quadratic form of the quadratic space V.

QuadraticNorm(v)

> The value $Q(v)$, where Q is the quadratic form attached to the generic space of the parent of the vector v.

QuadraticFormPolynomial(V)

> The polynomial $\sum_{i \leq j} q_{ij} x_i x_j$, where $Q = (q_{ij})$ is the matrix of the quadratic form of the quadratic space V.

QuadraticFormPolynomial(Q)

> The polynomial $\sum_{i \leq j} q_{ij} x_i x_j$, where $Q = (q_{ij})$ is an upper triangular matrix representing a quadratic form.

Example H29E12_____

The quadratic space defined by a polynomial.

```
> _<x,y,z> := PolynomialRing(Rationals(),3);
> f := x^2 + x*y +3*x*z - 2*y*z + y^2 +z^2;
> V := QuadraticSpace(f);
> PolarSpaceType(V);
quadratic space
> IsNonsingular(V);
true
> QuadraticFormMatrix(V);
[ 1  1  3]
[ 0  1 -2]
[ 0  0  1]
```

OrthogonalSum(V, W)

> The orthogonal direct sum V of the quadratic spaces V and W together with linear embeddings $V \to V \perp W$ and $W \to V \perp W$.

OrthogonalTensorProduct(V, W)

> The tensor product of the quadratic spaces V and W. If β and γ are the symmetric bilinear forms on V and W, respectively, and if ξ is the bilinear form on the tensor product, then $\xi(v_1 \otimes w_1, v_2 \otimes w_2) = \beta(v_1, v_2)\gamma(w_1, w_2)$.

TotallySingularComplement(V, U, W)

> Given totally singular subspaces U and W of the quadratic space V such that $U^\perp \cap W = 0$ this function returns a totally singular subspace X such that $V = X \oplus U^\perp$ and $W \subseteq X$.

Discriminant(V)

> If V is a vector space over the finite field K and if J is the Gram matrix of V, the *discriminant* of V is the determinant Δ of J modulo squares. That is, the discriminant is 0 if Δ is a square in K, 1 if it is a non-square. The form J is required to be non-degenerate.

ArfInvariant(V)

The Arf invariant of the quadratic space V.

Currently this is available only for quadratic spaces of even dimension $2m$ over a finite field F of characteristic 2. In this case there are two possibilities: either the Witt index of the form is m and the Arf invariant is 0, or the Witt index is $m - 1$ and the Arf invariant is 1.

DicksonInvariant(V, f)

The Dickson invariant of the isometry f of the quadratic space V is the rank (mod 2) of $1 - f$. If the polar form of Q is non-degenerate, the Dickson invariant defines a homomorphism from the orthogonal group $O(V)$ onto the additive group of order 2.

SpinorNorm(V, f)

The spinor norm of the isometry f of the quadratic space V. This is the discriminant of the Wall form (Section 29.10) of f.

HyperbolicBasis(U, B, W)

Given complementary totally singular subspaces U and W of a quadratic space and a basis B for U, return a sequence of pairwise orthogonal hyperbolic pairs whose second components form a basis for W.

OrthogonalReflection(a)

The reflection determined by a non-singular vector of a quadratic space.

RootSequence(V, f)

Given a matrix f representing an isometry of the quadratic space V, return a sequence of vectors such that the product of the corresponding orthogonal reflections is f. The empty sequence is returned if f is the identity matrix.

ReflectionFactors(V, f)

Given a matrix f representing an isometry of the quadratic space V, return a sequence of reflections whose product is f. The empty sequence corresponds to the identity matrix.

Given a quadratic space V defined by a quadratic form Q with polar form β and non-zero vectors $u, v \in V$ such that u is singular and $\beta(u, v) = 0$, the *Siegel transformation* (also called an Eichler transformation) is the isometry $\rho_{u,v}$ defined by

$$x\rho_{u,v} = x + \beta(x, v)u - \beta(x, u)v - Q(v)\beta(x, u)u.$$

SiegelTransformation(u, v)

The Siegel transformation defined by a singular vector u and a vector v orthogonal to u. The common parent of u and v must be a quadratic space.

Example H29E13_____

A group of isometries generated by Siegel transformations.

```
> Q := StandardQuadraticForm(4,3);
> V := QuadraticSpace(Q);
> u := V.1;
> QuadraticNorm(u);
0
> X := { v : v in V | DotProduct(u,v) eq 0 and QuadraticNorm(v) ne 0 };
> #X;
12
> H := sub< GL(V) | [SiegelTransformation(u,v) : v in X]>;
> #H;
9
```

29.7   Isometries and Similarities

If β is a bilinear or sesquilinear form on the vector space V, a linear transformation g of V is an *isometry* if g preserves β; it is a *similarity* if it preserves β up to a non-zero scalar multiple.

29.7.1   Isometries

IsIsometry(U, V, f)

> Returns `true` if the map f is an isometry from U to V with respect to the attached forms.

IsIsometry(f)

> Returns `true` if the map f is an isometry from its domain to its codomain.

IsIsometry(V, g)

> Returns `true` if the matrix g is an isometry of V with respect to the attached form.

IsIsometric(V, W)

> Determines whether the polar spaces V and W are isometric; if they are, an isometry is returned (as a map).

Example H29E14_____

A vector space is always equipped with a bilinear form and the default value is the identity matrix. However the "standard" symmetric form is $Q + Q^{\text{tr}}$, where Q is the standard quadratic form. In the following example the polar spaces defined by these forms are similar but not isometric.

```
> F := GF(5);
> V1 := VectorSpace(F,5);
> PolarSpaceType(V1);
orthogonal space
> WittIndex(V1);
2
> J2 := StandardSymmetricForm(5,F);
> J2;
[0 0 0 0 1]
[0 0 0 1 0]
[0 0 2 0 0]
[0 1 0 0 0]
[1 0 0 0 0]
> V2 := VectorSpace(F,5,J2);
> IsIsometric(V1,V2);
false
> V3 := VectorSpace(F,5,2*J2);
> flag, f := IsIsometric(V1,V3); flag;
true
> IsIsometry(f);
true
```

If J is an $n \times n$ matrix which represents a bilinear form and if M is a non-singular $n \times n$ matrix, then J and MJM^{tr} are said to be *congruent* and they define isometric polar spaces.

Conversely, given bilinear forms J_1 and J_2 the following example shows how to use the `IsIsometric` function to determine whether J_1 and J_2 are congruent and if so how to find a matrix M such that $J_1 = MJ_2M^{\text{tr}}$.

Example H29E15_____

Begin with alternating forms J_1 and J_2 over $\mathbf{F}_{25}$, construct the corresponding symplectic spaces and then use the isometry to define the matrix M.

```
> F<x> := GF(25);
> J1 := Matrix(F,4,4,[ 0, x^7, x^14, x^13, x^19, 0, x^8, x^5,
>   x^2, x^20, 0, x^17, x, x^17, x^5, 0 ]);
> J2 := Matrix(F,4,4,[ 0, x^17, 2, x^23, x^5, 0, x^15, x^5,
>   3, x^3, 0, 4, x^11, x^17, 1, 0 ]);
> V1 := SymplecticSpace(J1);
> V2 := SymplecticSpace(J2);
> flag, f := IsIsometric(V1,V2); assert flag;
```

```
> f;
Mapping from: ModTupFld: V1 to ModTupFld: V2 given by a rule
> M := Matrix(F,4,4,[f(V1.i) : i in [1..4]]);
> J1 eq M*J2*Transpose(M);
true
```

Example H29E16_____

Another way to obtain a matrix with the same effect as M is to use the function `TransformForm`.

```
> M1 := TransformForm(J1,"symplectic");
> M2 := TransformForm(J2,"symplectic");
> M_alt := M1*M2^-1;
> J1 eq M_alt*J2*Transpose(M_alt);
true
```

CommonComplement(V, U, W)

> A common complement to the subspaces U and W in the vector space V. (The subspaces must have the same dimension.) This is used by the following function, which implements Witt's theorem.

ExtendIsometry(V, U, f)

> An extension of the isometry $f : U \rightarrow V$ to an isometry $V \rightarrow V$, where U is a subspace of the polar space V.
>
> This is an implementation of Witt's theorem on the extension of an isometry defined on a subspace of a symplectic, unitary or quadratic space. The isometry f must satisfy $f(U \cap \mathrm{rad}(V)) = f(U) \cap \mathrm{rad}(V)$.
>
> If the characteristic is two and the form J of V is symmetric, then J must be alternating.

IsometryGroup(V)

> The group of isometries of the polar space V. This includes degenerate polar spaces as well as polar spaces defined by a quadratic form over a field of characteristic two.

Given a reflexive form J, the function `IsometryGroup(J)` defined in Chapter 92 returns the isometry group of J. More generally, if S is a sequence of reflexive forms, the function `IsometryGroup(S)` returns the group of isometries of the system.

Example H29E17 _____

We give an example of an isometry group of a degenerate quadratic space over a field of characteristic 2.

```
> F := GF(4);
> Q1 := StandardQuadraticForm(4,F : Minus);
> Q := DiagonalJoin(Q1,ZeroMatrix(F,2,2));
> V := QuadraticSpace(Q);
> G := IsometryGroup(V);
> [ IsIsometry(V,g) : g in Generators(G) ];
[ true, true, true, true, true, true, true ]
> #G;
96259276800
```

Example H29E18 _____

The matrix M constructed in Example H29E15 can be used to conjugate the isometry group of J_1 to the isometry group of J_2.

```
> F<x> := GF(25);
> J1 := Matrix(F,4,4,[ 0, x^7, x^14, x^13, x^19, 0, x^8, x^5,
>   x^2, x^20, 0, x^17, x, x^17, x^5, 0 ]);
> J2 := Matrix(F,4,4,[ 0, x^17, 2, x^23, x^5, 0, x^15, x^5,
>   3, x^3, 0, 4, x^11, x^17, 1, 0 ]);
> V1 := SymplecticSpace(J1);
> V2 := SymplecticSpace(J2);
> flag, f := IsIsometric(V1,V2); assert flag;
> M := Matrix(F,4,4,[f(V1.i) : i in [1..4]]);
> G1 := IsometryGroup(V1);
> G2 := IsometryGroup(V2);
> M^-1*G1.1*M in G2;
true
> M^-1*G1.2*M in G2;
true
```

29.7.2   Similarities

If β is a bilinear or sesquilinear form, a linear transformation f is a *similarity* with *multiplier* λ if $\beta(uf, vf) = \lambda\beta(u, v)$ for all u, v.

> IsSimilarity(U, V, f)

> Returns `true` if the map f is a similarity from U to V with respect to the attached forms. The second return value is the multiplier.

IsSimilarity(f)

> Returns `true` if the map f is a similarity from its domain to its codomain. The second return value is the multiplier.

IsSimilarity(V, g)

> Returns `true` if the matrix g is a similarity of V with respect to the attached form. The second return value is the multiplier.

IsSimilar(V, W)

> Determines whether the polar spaces V and W are similar; if they are, a similarity is returned (as a map).

Example H29E19 _____

An example of two unitary spaces, the first defined by an hermitian form, the second by a skew-hermitian form. The spaces are similar but not isometric.

```
> F<z> := GF(25);
> sigma := hom< F -> F | x :-> x^5 >;
> J1 := Matrix(F,4,4,[
>    0, z^3, z^14, z^9,   z^15, 2, z^21, z^5,
>    z^22, z^9, 1, z^7,   z^21, z, z^11, 4]);
> J2 := Matrix(F,4,4,[
>    z^15, z^10, z^17, z^7,  z^14, z^15, z^14, z^9,
>    z, z^10, z^3, z^20,     z^23, z^9, z^16, z^21]);
> V1 := UnitarySpace(J1,sigma);
> V2 := UnitarySpace(J2,sigma);
> IsUnitarySpace(V1);
true 1
> IsUnitarySpace(V2);
true -1
> IsIsometric(V1,V2);
false
> flag, f := IsSimilar(V1,V2);
> flag;
true
> IsSimilarity(V1,V2,f);
true z^3
```

SimilarityGroup(V)

> The group of similarities of the polar space V. This includes degenerate polar spaces as well as polar spaces defined by a quadratic form over a field of characteristic two.

29.8 Classical Groups

Given the matrix F of a bilinear or sesquilinear form, the functions described in the previous section can be used to construct classical groups which preserve F or preserve F up to a scalar.

Example H29E20_____

For example the following code constructs the symplectic group G and the conformal symplectic group C preserving the non-degenerate alternating form F.

```
> F := Matrix(GF(5),4,4,[0,-1,-1,1, 1,0,1,-1, 1,-1,0,1, -1,1,-1,0]);
> V := SymplecticSpace(F);
> G := IsometryGroup(V);
> C := SimilarityGroup(V);
> f1, _ := IsIsomorphic(G,Sp(4,5));
> f2, _ := IsIsomorphic(C,CSp(4,5));
> f1, f2;
true true
```

If only the symplectic group is needed this can be obtained on one line

```
> G := IsometryGroup(SymplecticSpace(F));
```

_____

Similar code can be used to construct the orthogonal group and the conformal orthogonal group preserving a quadratic form or the unitary group and conformal unitary group preserving an hermitian form.

However, to construct a group strictly between the isometry group and the similarity group of a form some more work is needed. The following example illustrates how this may be carried out for an hermitian form.

Example H29E21_____

This example constructs the group of unitary matrices of determinant 1 which preserve an hermitian form F. The first step is to define a MAGMA function which takes the form as its only argument. The base field and dimension of F can be recovered from F itself and then an isometry φ is constructed from the unitary space of the standard form to the unitary space of F. This is used to transform the generators of a standard copy of the special unitary group to the group preserving the form.

```
> specialUnitaryGrp := function(F)
>   K := BaseRing(F); n := NumberOfRows(F);
>   J, sigma := StandardHermitianForm(n,K);
>   V := UnitarySpace(F,sigma);
>   S := UnitarySpace(J,sigma);
>   flag, phi := IsIsometric(S,V);
>   assert flag;
>   T := Matrix(K,n,n,[phi(S.i) : i in [1..n]]);
>   H := SpecialUnitaryGroup(n,K);
>   return sub<IsometryGroup(V) | [T^-1*H.i*T : i in [1..NumberOfGenerators(H)]]>;
```

```
> end function;
```

To complete the example we define an hermitian form F and then use the above function to find the special unitary group that preserves F.

```
> K<z> := GF(9);
> F := Matrix(K,4,4,[1,z^7,z^6,2, z^5,2,z^6,z^6,  z^2,z^2,1,z^3, 2,z^2,z,0]);
> G := specialUnitaryGrp(F);
> flag where flag is IsIsomorphic(G,SU(4,3));
true
```

29.9   Lie Algebras and Bilinear Forms

If J is the matrix of a bilinear form, the Lie algebra of derivations of J consists of the matrices X such that $XJ + JX^{\mathrm{tr}} = 0$.

DerivationAlgebra(J)		

Rep	MonStgElt	*Default* : "Sparse"
Check	BoolElt	*Default* : false

  The Lie algebra of derivations of the bilinear form with matrix J. The possible values for Rep are "Dense", "Sparse" and "Partial" with the default being "Sparse".

Example H29E22_____

Construct the Lie algebra preserving the standard alternating form of rank 6 over the field of 7 elements and check that it is a simple algebra of type C_3.

```
> J := StandardAlternatingForm(6,7);
> L := DerivationAlgebra(J);
> IsSimple(L);
true
> SemisimpleType(L);
C3
```

Example H29E23_____

Over a field of characteristic 2, the Lie algebra of an alternating form is no longer simple. In this example the Lie algebra L of the standard alternating form of rank 6 over $\mathbf{F}_8$ is of symplectic type C_3 but has an ideal I of type G_2. The dimension of the centre of L is 1 and so the ideal I will appear as either the first or second composition factor.

```
> J := StandardAlternatingForm(6,8);
> L := DerivationAlgebra(J);
> SemisimpleType(L);
C3
> Dimension(Centre(L));
1
```

```
> CF := CompositionFactors(L);
> CF;
[
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 14 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3),
    Lie Algebra of dimension 1 with base ring GF(2^3)
]
> exists(I){I : I in CF | Dimension(I) eq 14 };
true
> IsSimple(I);
true
> SemisimpleType(I);
G2
```

Another way to construct a Lie algebra from an alternating form β with matrix J defined on a vector space V of dimension n over a field F is to set $L = V \oplus F$ and define the multiplication by $[e_i, e_j] = \beta(e_i, e_j)$, where $e_1, e_2, \ldots, e_n$ is a basis for V. (All other structure constants are 0.) This is the (generalised) Heisenberg algebra.

HeisenbergAlgebra(J)		
Rep	MONSTGELT	*Default :* "Sparse"
Check	BOOLELT	*Default :* false

The nilpotent Lie algebra whose structure constants are obtained from the alternating form with matrix J as described above. The possible values for Rep are `"Dense"`, `"Sparse"` and `"Partial"` with the default being `"Sparse"`.

Example H29E24_____

Continuing the previous example we construct an explicit isomorphism between the quotient of L by its G_2 ideal and the Heisenberg algebra of the form.

```
> Q := CS[1];
> W := L/Q;
> W;
Lie Algebra of dimension 7 with base ring GF(2^3)
> Z := Centre(W);
> z := Z.1;
> exists(u1,v1){ <u,v> : u,v in W | u*v ne 0 and u*v eq z };
true
> W1 := Centraliser(W,sub<W|u1,v1>);
true
```

```
> exists(u2,v2){ <u,v> : u,v in W1 | u*v ne 0 and u*v eq z };
true
> W2:= Centraliser(W1,sub<W1|u2,v2>);
> exists(u3,v3){ <u,v> : u,v in W2 | u*v ne 0 and u*v eq z };
> H := HeisenbergAlgebra(J);
> f := hom< H -> W | u1,u2,u3,v3,v2,v1,z >;
> forall{ <u,v> : u,v in Basis(H) | f(u*v) eq f(u)*f(v) };
true
> Kernel(f);
Lie Algebra of dimension 0 with base ring GF(2^3)
> Image(f) eq W;
true
```

29.10   Wall Forms

Given an isometry f of a quadratic, symplectic or unitary space V with bilinear or sesquilinear form β, the Wall form of f is the form θ defined on the image I of $1 - f$ by $\theta(u,v) = \beta(w,v)$, where $u = w(1-f)$. In general, the Wall form is not reflexive.

WallForm(V, f)

> The space of the Wall form of the isometry f and its embedding in V.

WallIsometry(V, I, mu)

> The inverse of WallForm. This is an isometry corresponding to the embedding $\mu : I \to V$, where V is a quadratic, symplectic or unitary space.

WallDecomposition(V, f)

> An isometry f of a quadratic or symplectic space V is *Wall-regular* if the restriction of $1 - f$ to the image of $1 - f$ is invertible. If f is any isometry of V this function returns a Wall-regular element f_r and a unipotent element f_u such that $f = f_r f_u = f_u f_r$.

SemiOrthogonalBasis(V)

> If V is a vector space with a bilinear form β, a basis e_1, e_2, ..., e_n for V is semi-orthogonal if $\beta(e_i, e_j) = 0$ for $i < j$. This function returns a semi-orthogonal basis with respect to the non-degenerate, non-alternating form attached to V. If the base field is $\mathbf{F}_2$, the form should be symmetric.

GeneralisedWallForm(V, f)

> The space of the generalised Wall form of the similarity f and its embedding in V.

29.11   Invariant Forms

Given a group G which acts on a vector space V over a finite field F, the space of all G-invariant bilinear forms is isomorphic to $\mathrm{Hom}_G(V, V^*)$, where V^* is the dual space of V. The isomorphism associates the form β to $\theta \in \mathrm{Hom}_G(V, V^*)$, where $\beta(u, v) = \langle v, u\theta \rangle$ and where $\langle v, \varphi \rangle$ denotes the action of φ on v. If $v_1, v_2, \ldots, v_n$ is a basis for V with dual basis $\omega_1, \omega_2, \ldots, \omega_n$, the matrix of θ with respect to these bases is $J = (\beta(e_i, e_j))$.

A linear transformation with matrix A preserves the form if and only if $AJA^{\mathrm{tr}} = J$.

If the characteristic of the field is not 2, then $J = \frac{1}{2}(J + J^{\mathrm{tr}}) + \frac{1}{2}(J - J^{\mathrm{tr}})$. Therefore, in this case, every G-invariant form is the sum of a G-invariant symmetric form and a G-invariant alternating form. If the characteristic of the field is 2, every alternating form is symmetric. Thus in this case the space of G-invariant alternating forms is a subspace of the space of G-invariant symmetric forms. If the G-module is irreducible these two spaces coincide.

`InvariantBilinearForms(G)`

> Given a matrix group G this function returns two sequences: a basis for the space of G-invariant symmetric forms and a basis for the space of G-invariant alternating forms.

Example H29E25 _____

In this example the group G is reducible but (up to a scalar multiple) there is a unique G-invariant bilinear form.

```
> F<x> := GF(25);
> G := MatrixGroup< 4, F |
>    [ 1, 0, 0, 0,  0, 1, 0, 0,  0, x^14, 1, 0,  0, 0, 0, 1 ],
>    [ 3, x^23, x^20, x^10,  2, 3, 0, x^13,  4, x^10, x^13, x^23,
>      x^5, x^11, x, x^17 ] >;
> IsIrreducible(G);
false
> InvariantBilinearForms(G);
[]
[
    [    0    0    0    1]
    [    0    0    1    0]
    [    0    4    0    0]
    [    4    0    0    0]
]
```

If G acts irreducibly on a vector space V of dimension n over a (finite) field F and if $\theta_0 : V \to V^*$ is a G-invariant isomorphism, then $D \to \mathrm{Hom}_G(V, V^*) : \theta \mapsto \theta\theta_0$ is an isomorphism of vector spaces, where $D = \mathrm{End}_G(V)$. The algebra D is a division ring and hence a field (since F is finite). Thus V becomes a vector space of dimension m over D, where $n = m|D : F|$ and G is isomorphic to a subgroup of $\mathrm{GL}(m, D)$.

Example H29E26

If G acts irreducibly on V, the spaces of symmetric and alternating G-invariant forms are isomorphic (as vector spaces) to subfields of $\mathrm{End}_G(V)$ and therefore their dimensions are either 0 or divide $\dim_F(V)$.

```
> F<a> := GF(25);
> G := MatrixGroup< 4, F |
>    [ a^10, a^21, a^4, 4,
>      a^16, 4, a^9, a^8,
>      a^20, 4, 4, a^13,
>      0, a^2, a^11, a ] >;
> IsIrreducible(G), #G;
true 626
> sym, alt := InvariantBilinearForms(G);
> #sym,#alt;
2 2
```

If the characteristic of the field is not two and if J is a symmetric bilinear form there is a unique upper triangular matrix Q such that $J = Q + Q^{\mathrm{tr}}$.

On the other hand, if the characteristic is two and J is alternating, the upper triangular matrices Q such that $J = Q + Q^{\mathrm{tr}}$ form an affine space of dimension $\dim V$.

Suppose that the characteristic is two. If G preserves a symmetric bilinear form which is not alternating, then G is reducible. Conversely, if G is irreducible and if J is the matrix of a symmetric form preserved by G, then the form must be alternating and there is a unique G-invariant quadratic form Q such that $J = Q + Q^{\mathrm{tr}}$.

| InvariantQuadraticForms(G) |

A basis for the space of quadratic forms preserved by the irreducible matrix group G.

Example H29E27

In the following example the quadratic forms which are invariant under the action of a cyclic group H of order 13 form a vector space of dimension 3 over $\mathbf{F}_4$.

```
> F<z> := GF(4);
> H := MatrixGroup<6,F |
>     [ z, 0, z^2, z, z, 1,
>       1, z, 0, z, z, z,
>       0, z^2, z, 1, z^2, z^2,
>       z, 1, z, 1, 1, 0,
>       1, z^2, z, z, 0, 1,
>       1, 0, 1, 0, z^2, 1 ] >;
>
> InvariantQuadraticForms(H);
[
    [ 1   1   0   1   0   0]
    [ 0   0   1 z^2 z^2   z]
```

```
[ 0   0   1   0 z^2    z]
[ 0   0   0   0   0    z]
[ 0   0   0   0 z^2    z]
[ 0   0   0   0   0 z^2],

[ 1   0   1 z^2   1    0]
[ 0   z   1   1   1    z]
[ 0   0   z   0   1    0]
[ 0   0   0 z^2 z^2 z^2]
[ 0   0   0   0 z^2    1]
[ 0   0   0   0   0    1],

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

Given a group G which acts on a vector space V over a finite field F with an automorphism $F \to F : a \mapsto \overline{a}$ of order 2, the space of G-invariant sesquilinear forms is isomorphic to the space of G-invariant semilinear maps from V to V^*; equivalently it is isomorphic to $\mathrm{Hom}_G(V, \overline{V}^*)$, where $\overline{V}^*$ is the semilinear dual of V, namely the space of all semilinear maps from V to F.

If $\theta \in \mathrm{Hom}_G(V, \overline{V}^*)$, the corresponding sesquilinear form β is defined by $\beta(u,v) = \langle v, u\theta \rangle$ where, as before, $\langle v, \varphi \rangle$ denotes the action of φ on v.

> **SemilinearDual(M, mu)**

The semilinear dual of the G-module M with respect to the field automorphism `mu`.

> **InvariantSesquilinearForms(G)**

A basis for the space of *hermitian* forms preserved by the matrix group G.

Example H29E28_____

Let F_0 be the fixed field of the involution. The set $\mathcal{H}$ of G-invariant hermitian forms is a vector space over F_0 and if the characteristic of F is not 2, then $\mathrm{Hom}_G(V, \overline{V}^*) \simeq \mathcal{H} \otimes_{F_0} F$.

```
> F<x> := GF(5,2);
> mu := hom< F->F | x :-> x^5 >;
> H := MatrixGroup< 5, F |
>    [ 0, x^3, 0, 1, x^9, x^8, 1, 0, x^11, x^7, x^20, x^16, 1,
>     x^11, x^3, x^21, 4, 1, x^3, x^23, x^4, x^3, x, x^3, 2 ] >;
> M := GModule(H);
> D := SemilinearDual(M,mu);
> E := AHom(M,D);
> Dimension(E);
```

```
5
> herm := InvariantSesquilinearForms(H);
> #herm;
5
```

Example H29E29_____

If an irreducible group preserves both a bilinear and a sesquilinear form then it is realisable over a subfield of its base field. Conversely, this observation can be used to construct an example:

```
> F<x> := GF(81);
> H := MatrixGroup< 4, F | [ChangeRing(g,F) : g in Generators(Sp(4,9))]>;
> InvariantBilinearForms(H);
[]
[
    [   0    0    0    1]
    [   0    0    1    0]
    [   0    2    0    0]
    [   2    0    0    0]
]
> InvariantSesquilinearForms(H);
[
    [   0    0    0 x^45]
    [   0    0 x^45    0]
    [   0  x^5    0    0]
    [ x^5    0    0    0]
]
```

InvariantFormBases(G)

> This function returns four sequences: bases for the spaces of symmetric, alternating, hermitian and quadratic forms preserved by the matrix group G.

29.11.1   Semi-invariant Forms

Given a vector space V over a finite field F and a group G which acts on V, a bilinear form $\beta : V \times V \to F$ is *semi-invariant* if for all $g \in G$ there is a scalar $\lambda(g)$ such that $\beta(ug, vg) = \lambda(g)\beta(u, v)$ for all $u, v \in V$. The function $\lambda : G \to F^\times$ is a homomorphism and its kernel contains the derived group of G. The *twisted dual* V_λ^* of the G-module V is the dual space of V with G-action given by $\langle v, \varphi g \rangle = \lambda(g)\langle vg^{-1}, \varphi \rangle$; thus if A is the matrix of g acting on V the matrix of the action on V_λ^* with respect to the dual basis is $\lambda(g)A^{-\mathrm{tr}}$.

The space of all semi-invariant bilinear forms is isomorphic to $\mathrm{Hom}_G(V, V_\lambda^*)$. The isomorphism associates the form β to $\theta \in \mathrm{Hom}_G(V, V_\lambda^*)$, where $\beta(u, v) = \langle v, u\theta \rangle$.

If β is a bilinear form with matrix J, then the linear transformation g with matrix A preserves the form up to multiplication by $\lambda(g)$ if and only if $AJA^{\mathrm{tr}} = \lambda(g)J$.

> **TwistedDual(M, lambda)**

The twisted dual of the *G*-module M with respect to the linear character `lambda`.

> **SemiInvariantBilinearForms(G)**

A sequence of triples $\langle L, S, A \rangle$ where L is a sequence of field elements (one for each generator) which define a homomorphism from the matrix group G to its base field and S and A are bases for the spaces of symmetric and alternating forms preserved by G (up to multiplication by scalars).

> **SemiInvariantQuadraticForms(G)**

A sequence of pairs $\langle L, Q \rangle$ where L is a sequence of field elements (one for each generator) which define a homomorphism from the matrix group G to its base field and Q is a basis for the space of quadratic forms preserved by G (up to multiplication by scalars).

> **TwistedSemilinearDual(M, lambda, mu)**

The twisted semilinear dual of the *G*-module M with respect to the linear character λ and the field automorphism μ.

> **SemiInvariantSesquilinearForms(G)**

A sequence of pairs $\langle L, H \rangle$ where L is a sequence of field elements (one for each generator) which define a homomorphism from the matrix group G to the field F_0, where the base field of G is a quadratic extension of F_0, and H is a basis for the space of hermitian forms preserved by G (up to multiplication by scalars).

Example H29E30_____

In this example H is a normal subgroup of the absolutely irreducible group N an H is irreducible but not absolutely irreducible.

```
> F<x> := GF(3,2);
> H := MatrixGroup<3,F|
>   [x^2,x^7,x^3, x,0,1, x^3,x^6,2],
>   [x^3, 0, 0, 0, x^3, 0, 0, 0, x^3 ] >;
> N := MatrixGroup<3,F|H.1,H.2,[x^5,x^5,2, 0,x^2,x^6, x^7,x^7,2]>;
> IsNormal(N,H);
true
> IsIrreducible(H), IsAbsolutelyIrreducible(H);
true false
> IsIrreducible(N), IsAbsolutelyIrreducible(N);
true true
> SemiInvariantSesquilinearForms(H);
[
    <[ 1, 2 ],
    [
        [  1   x   x]
        [x^3   0 x^3]
        [x^3   x   1],
```

```
        [  0 x^3   x]
        [  x   0 x^5]
        [x^3 x^7   0],

        [  0   0   1]
        [  0   1   0]
        [  1   0   0]
    ]>
]
> SemiInvariantSesquilinearForms(N);
[
    <[ 1, 2, 1 ],
    [
        [  0   0   1]
        [  0   1   0]
        [  1   0   0]
    ]>
]
```

29.12   Bibliography

[**Bou07**] N. Bourbaki. *Éléments de mathématique. Algèbre. Chapitre 9.* Springer-Verlag, Berlin, 2007. Reprint of the 1959 original.

[**Bra36**] Richard Brauer. A characterization of null systems in projective space. *Bull. Amer. Math. Soc.*, 42(4):247–254, 1936.

[**Bue95**] Francis Buekenhout. *Handbook of incidence geometry.* North-Holland, Amsterdam, 1995.

[**Tay92**] Donald E. Taylor. *The geometry of the classical groups*, volume 9 of *Sigma Series in Pure Mathematics*. Heldermann Verlag, Berlin, 1992.