Handbook of Magma Functions

Volume 9

Commutative Algebra and Algebraic Geometry

John Cannon Wieb Bosma

Claus Fieker Allan Steel

Editors

Version 2.19

Sydney

December 17, 2012

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 9: OVERVIEW

VOLUME 9: CONTENTS

XV ALGEBRAIC GEOMETRY 3465

PART XIV
COMMUTATIVE ALGEBRA

105 GRÖBNER BASES

Chapter 105
GRÖBNER BASES

105.1 Introduction

This chapter describes the basics for configuring Magma's powerful *Gröbner basis* machinery, which lies at the heart of computations with ideals and modules over multivariate polynomial rings. Later chapters will describe the many functions and operations available to the user for working with ideal and modules.

Gröbner bases were introduced by Bruno Buchberger [Buc65] and at the heart of the theory is the *Buchberger algorithm* which computes a Gröbner basis of an ideal starting from an arbitrary basis (generating set) of the ideal. The two books *Ideals, Varieties and Algorithms* [CLO96] and *Gröbner Bases* [BW93] have also inspired much of the design and presentation of ideals of multivariate polynomial rings in Magma.

Since V2.11 (May 2004), Magma also contains a highly optimized implementation of the Faugère F_4 algorithm [Fau99], based on sparse linear algebra techniques, which usually performs dramatically better than the Buchberger algorithm (see [Ste04]).

Chapter 24 deals with the basics of multivariate polynomial rings and their elements (for which there are very many functions), so it is recommended that that chapter be perused before reading this one.

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 110. See also the chapters on affine algebras (Chapter 108) and on modules over affine algebras (Chapter 109), and the chapter on algebraically closed fields (Chapter 40), which allows one to compute the variety of an ideal over the algebraic closure of the base field.

105.2 Representation and Monomial Orders

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 (or indeterminates) 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.

Monomial orders are of critical importance when dealing with Gröbner bases. Let M be the set of all monomials of P. A *monomial ordering* on M is a total order $<$ on M such that $1 \leq s$ for all $s \in M$, $s \leq t$ implies $su \leq tu$ for all $s, t, u \in M$, and M is a well-ordering (every non-empty subset of M possesses a minimal element w.r.t. $<$). Monomial orders can be naturally specified in terms of *weight vectors*: a vector W from $\mathbf{Q}^n$ with non-negative entries is called a weight vector since it weights a monomial s by the product $s.W$ (defined to be the dot product of the exponent vector of s with W); any sequence of

n linearly-independent weight vectors determines a monomial order on M (see the weight order below [subsection 105.2.9]). All monomial orderings can in fact be represented in terms of weight vectors.

Multivariate polynomial rings are constructed in Magma such that the monomials of any polynomial are sorted with respect to a specified monomial order, with the greatest monomial first. Gröbner basis computations are dramatically affected by the choice of monomial order. Magma provides an extensive choice of monomial orders. Currently, the intrinsic functions `PolynomialRing` (or `PolynomialAlgebra`), `ChangeOrder` and `VariableExtension` expect a monomial order; it is specified by a string giving the name, optionally followed by extra arguments for that order.

We now describe each of the monomial orders available in Magma. We suppose that s and t are monomials from P which has rank n. Any order on the monomials is then fully defined by just specifying exactly when $s < t$ with respect to that order. In the following, the argument(s) are described for an order as a list of expressions; that means that the expressions (without the parentheses) should be appended to any base arguments when any particular intrinsic function is called which expects a monomial order. See also [CLO96, Chap. 2, §2] for more details about the first three orders.

105.2.1 Lexicographical: `lex`

Definition: $s < t$ iff there exists $1 \leq i \leq n$ such that the first $i-1$ exponents of s and t are equal but the i-th exponent of s is less than the i-th exponent of t. The order is specified by the argument (`"lex"`).

The order is called "lexicographical" since it orders the monomials as if they were words in a dictionary. The i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. A Gröbner basis of an ideal with respect to the lexicographical order usually represents the most information about the ideal but can be hard to compute.

105.2.2 Graded Lexicographical: `glex`

Definition: $s < t$ iff the total degree of s is less than the total degree of t or the total degree of s is equal to the total degree of t and $s < t$ with respect to the lexicographical order. The order is specified by the argument (`"glex"`).

The order is called "graded lexicographical" since it first grades the monomials by total degree, and then decides ties by the lexicographical order. The i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. This order is rarely used because the `grevlex` order below is usually a better degree order (i.e., yields smaller Gröbner bases).

105.2.3 Graded Reverse Lexicographical: `grevlex`

Definition: $s < t$ iff the total degree of s is less than the total degree of t or the total degree of s is equal to the total degree of t and $s > t$ with respect to the lexicographical order applied to the exponents of s and t in reverse order. The order is specified by the argument (`"grevlex"`).

The order is called “graded reverse lexicographical” since it first grades the monomials by total degree, and then decides ties by the negation of the lexicographical order applied to the variables in reverse order. Again, the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. A Gröbner basis of an ideal with respect to the graded reverse lexicographical order is usually the easiest to compute so it is recommended that this order be used when just any Gröbner basis for an ideal is desired.

105.2.4 Graded Reverse Lexicographical (Weighted): grevlexw

Definition (given a sequence W of n positive integer weights): $s < t$ iff the total weighted degree d_s of s w.r.t. W is less than the total degree d_t of t w.r.t. W or $d_s = d_t$ and $s > t$ with respect to the lexicographical order applied to the exponents of s and t in reverse order. The order is specified by the arguments `("grevlexw", W)`.

The order is called “graded reverse lexicographical (weighted)” since it first grades the monomials by weighted degree w.r.t. W, and then decides ties by the negation of the lexicographical order applied to the variables in reverse order. If $W = [1, 1, \dots, 1]$, then this order is equal to the `grevlex` order. Again, the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable.

This order is similar to the `grevlex` order, but is useful if an ideal is homogeneous with respect to the grading given by W, since the Gröbner basis of the ideal will tend to be smaller with this order.

105.2.5 Elimination (k): elim

Definition (given k with $1 \leq k \leq n-1$): $s < t$ iff $s_k < t_k$ with respect to the `grevlex` order or $s_k = t_k$ and $s_{k'} < t_{k'}$ with respect to the `grevlex` order where m_k denotes the monomial consisting of the first k exponents of m and $m_{k'}$ denotes the monomial consisting of the last $n-k$ exponents of m (this order is thus the concatenation of two block `grevlex` orders). The order is specified by the arguments `("elim", k)`.

The order is called “elimination” since the first k variables are “eliminated”: if G is a Gröbner basis of an ideal I of the polynomial ring $K[x_1, \dots, x_n]$ with respect to this order, then $G \cap K[x_{k+1}, \dots, x_n]$ is a Gröbner basis of the k-th elimination ideal $I \cap K[x_{k+1}, \dots, x_n]$. (It is usually easier to compute a Gröbner basis with respect to this order for any k than with respect to the full lexicographical order.) Again, the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable.

105.2.6 Elimination List: elim

Definition (given sequences U and V such that U and V contain distinct integers in the range 1 to n and the sum of the lengths of U and V is n and U and V are disjoint): $s < t$ iff $s_U < t_U$ with respect to the `grevlex` order or $s_U = t_U$ and $s_V < t_V$ with respect to the `grevlex` order where m_L denotes the monomial consisting of the exponents of m corresponding to the entries of L in order. The order is specified by the arguments `("elim", U, V)`. V may be omitted if desired so the arguments are just `("elim", U)`; in this case V is chosen to be an appropriate sequence to complement U.

The order is called "elimination" since the variables in U are "eliminated". The order of the elements in U and V are significant since the ordering on the variables makes $U[1]$ greatest, then $U[2]$, etc., then $V[1]$, $V[2]$, etc.

105.2.7 Inverse Block: `invblock`

Definition (given sequences U and V such that U and V contain distinct integers in the range 1 to n and the sum of the lengths of U and V is n and U and V are disjoint): $s < t$ iff $s_V < t_V$ with respect to the `grevlex` order or $s_V = t_V$ and $s_U < t_U$ with respect to the `grevlex` order. The order is specified by the arguments (`"invblock", U, V`). V may be omitted if desired so the arguments are just (`"invblock", U`); in this case V is chosen to be an appropriate sequence to complement U.

The order is called "inverse block" since it applies a block ordering on the exponents on V then U which inverts the lists supplied to the elimination list order. Thus this is the same as the elimination order except that the lists U and V are swapped. See [BW93, p. 390] for the motivation for this order.

105.2.8 Univariate: `univ`

Definition (given i with $1 \leq i \leq n$): $s < t$ iff $s_L < t_L$ with respect to the `grevlex` order or $s_L = t_L$ and the i-th exponent of s is less than the i-th exponent of t, where L is the sequence `[1 .. n]` with i deleted. The order is specified by the arguments (`"univ", i`).

The order is called "univariate" since when monomials are compared, any monomial not containing the i-th variable is greater than any monomial containing the i-th variable. Thus all variables but the i-th are "eliminated" so that a Gröbner basis of a zero-dimensional ideal I with this ordering will contain the unique monic generator of the elimination ideal consisting of all the polynomials in I containing the i-th variable alone. The j-th variable is greater than the $(j + 1)$-th variable for $1 \leq j < i$ and $i < j \leq n$ and the j-th variable is greater than the i-th variable for any $j \neq i$.

105.2.9 Weight: `weight`

Definition (given n weight vectors $W_1, \ldots W_n$ from $\mathbf{Q}^n$): $s < t$ iff there exists $1 \leq i \leq n$ such that $s.W_j = t.W_j$ for $1 \leq j < i$ and $s.W_i < t.W_i$. The order is specified by the arguments (`"weight", Q`) where Q is a sequence of n^2 non-negative integers or rationals describing the n weight vectors of length n (in row major order).

The n weight vectors must describe a vector space basis of $\mathbf{Q}^n$ (i.e., be linearly-independent), since otherwise this would not yield a total ordering on the monomials. The weight order allows one to specify any possible monomial order; any of the monomial orders mentioned above can be specified by an appropriate choice of weight vectors. However, using the in-built versions of the specialized orders above is much faster than constructing versions of them based on weight vectors. The next section contains an example in which a polynomial ring is constructed with a weight order for the monomials.

105.3 Polynomial Rings and Ideals

105.3.1 Creation of Polynomial Rings and Accessing their Monomial Orders

Multivariate polynomial rings are created from a coefficient ring, the number of variables, and a monomial order. If no order is specified, the monomial order is taken to be the lexicographical order. This section is briefly repeated from the section 24.2.1 in the multivariate polynomial rings chapter, so as to show how one can set up the polynomial ring in which to create an ideal.

Please note that the Gröbner basis of an ideal with respect to the lexicographical order is often much more complicated and difficult to compute than the Gröbner basis of the same ideal with respect to other monomial orders (e.g. the `grevlex` order), so it may be preferable to use another order if the Gröbner basis with respect to any order is desired (see also the function `EasyIdeal` below). Yet the lexicographical order is the most natural order and is often the desired order so that is why it is used by default if no specific order is given.

`PolynomialRing(R, n)`

`PolynomialAlgebra(R, n)`

`Global` BoolElt *Default :* `false`

Create a multivariate polynomial ring in $n > 0$ variables 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 next function).

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 $n > 0$ variables over the ring R with the given order *order* on the monomials. See the section on monomial orders for the valid values for the argument *order*.

`PolynomialRing(R, n, T)`

`PolynomialAlgebra(R, n, T)`

Create a multivariate polynomial ring in $n > 0$ variables over the ring R with the order given by the tuple T on the monomials. T must be a tuple whose components match the valid arguments for the monomial orders in Section 105.2 (or a tuple returned by the following function `MonomialOrder`).

`MonomialOrder(P)`

Given a polynomial ring P (or an ideal thereof), return a description of the monomial order of P. This is returned as a tuple which matches the relevant arguments listed for each possible order in Section 105.2, so may be passed as the third argument to the function `PolynomialRing` above.

`MonomialOrderWeightVectors(P)`

Given a polynomial ring P of rank n (or an ideal thereof), return the weight vectors of the underlying monomial order as a sequence of n sequences of n rationals. See, for example, [CLO98, p. 153] for more information.

Example H105E1

We show how one can construct different polynomial rings with different orders.

```
> Z := IntegerRing();
> // Construct polynomial ring with DEFAULT lex order
> P<a,b,c,d> := PolynomialRing(Z, 4);
> MonomialOrder(P);
<"lex">
> MonomialOrderWeightVectors(P);
[
    [ 1, 0, 0, 0 ],
    [ 0, 1, 0, 0 ],
    [ 0, 0, 1, 0 ],
    [ 0, 0, 0, 1 ]
]
> // Construct polynomial ring with grevlex order
> P<a,b,c,d> := PolynomialRing(Z, 4, "grevlex");
> MonomialOrder(P);
<"grevlex">
> MonomialOrderWeightVectors(P);
[
    [ 1, 1, 1, 1 ],
    [ 1, 1, 1, 0 ],
    [ 1, 1, 0, 0 ],
    [ 1, 0, 0, 0 ]
]
> // Construct polynomial ring with block elimination and a > d > b > c
> P<a,b,c,d> := PolynomialRing(Z, 4, "elim", [1, 4], [2, 3]);
> MonomialOrder(P);
<"elim", [ 1, 4 ], [ 2, 3 ]>
> MonomialOrderWeightVectors(P);
[
    [ 1, 0, 0, 1 ],
    [ 1, 0, 0, 0 ],
    [ 0, 1, 1, 0 ],
    [ 0, 1, 0, 0 ]
]
> a + b + c + d;
a + d + b + c
> a + d^10 + b + c^10;
d^10 + a + c^10 + b
> a + d^10 + b + c;
d^10 + a + b + c
> // Construct polynomial ring with weight order and x > y > z
> P<x, y, z> := PolynomialRing(Z, 3, "weight", [100,10,1, 1,10,100, 1,1,1]);
> MonomialOrder(P);
<"weight", [ 100, 10, 1, 1, 10, 100, 1, 1, 1 ]>
> MonomialOrderWeightVectors(P);
[
    [ 100, 10, 1 ],
    [ 1, 10, 100 ],
    [ 1, 1, 1 ]
]
> x + y + z;
x + y + z
> (x+y^2+z^3)^4;
x^4 + 4*x^3*y^2 + 4*x^3*z^3 + 6*x^2*y^4 + 12*x^2*y^2*z^3 +
    6*x^2*z^6 + 4*x*y^6 + 12*x*y^4*z^3 + 12*x*y^2*z^6 +
    4*x*z^9 + y^8 + 4*y^6*z^3 + 6*y^4*z^6 +
    4*y^2*z^9 + z^12
```

105.3.2 Creation of 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*. A polynomial of the ring whose monomials all have the same weighted degree is called *homogeneous*. The polynomial ring can be decomposed as the direct sum of graded homogeneous components.

Suppose a polynomial ring P has n variables $x_1, \ldots, x_n$ and the weights for the variables are $d_1, \ldots, d_n$ respectively. Then for a monomial $m = x_1^{e_1} \cdots x_n^{e_n}$ of P (with $e_i \geq 0$ for $1 \leq i \leq n$), the *weighted degree* of m is defined to be $\sum_{i=1}^{n} e_i d_i$.

A polynomial ring created without a specific weighting (using the default version of the `PolynomialRing` function or similar) has weight 1 for each variable so the weighted degree coincides with the total degree.

The following functions allow one to create and operate on elements of polynomial rings with specific weights for the variables.

```
PolynomialRing(R, Q)
```

```
PolynomialAlgebra(R, Q)
```

Given a ring R and a non-empty sequence Q of positive integers, create a multivariate polynomial ring in $n = \#Q$ variables over the ring R with the weighted degree of the i-th variable set to be $Q[i]$ for each i. The rank n of the polynomial is determined by the length of the sequence Q. (The angle bracket notation can be used to assign names to the variables, just like in the usual invocation of the `PolynomialRing` function.)

As of V2.15, the default monomial order chosen is the `grevlexw` order with weights given by Q, since the Gröbner basis of an ideal w.r.t. this order tends to be smaller if the ideal is homogeneous w.r.t. the grading.

```
Grading(P)
```

```
VariableWeights(P)
```

Given a graded polynomial ring P (or an ideal thereof), return the variable weights of P as a sequence of n integers where n is the rank of P. If P was constructed without specific weights, the sequence containing n copies of the integer 1 is returned.

105.3.3 Element Operations Using the Grading

```
Degree(f)
```

```
WeightedDegree(f)
```

Given a polynomial f of the graded polynomial ring P, this function returns the weighted degree of f, which is the maximum of the weighted degrees of all monomials that occur in f. The weighted degree of a monomial m depends on the weights assigned to the variables of the polynomial ring P — see the introduction of this section for details. Note that this is different from the natural total degree of f which ignores any weights.

```
LeadingWeightedDegree(f)
```

Given a polynomial f of the graded polynomial ring P, this function returns the leading weighted degree of f, which is the weighted degree of the leading monomial of f. The weighted degree of a monomial m depends on the weights assigned to the variables of the polynomial ring P — see the introduction of this section for details.

```
IsHomogeneous(f)
```

Given a polynomial f of the graded polynomial ring P, this function returns whether f is homogeneous with respect to the weights on the variables of P (i.e., whether the weighted degrees of the monomials of f are all equal).

```
HomogeneousComponent(f, d)
```

Given a polynomial f of the graded polynomial ring P, this function returns the *weighted* degree-d homogeneous component of f which is the sum of all the terms of f whose monomials have weighted degree d. d must be greater than or equal to 0. If f has no terms of weighted degree d, then the result is 0.

```
HomogeneousComponents(f)
```

Given a polynomial f of the graded polynomial ring P, this function returns the *weighted* degree-d homogeneous component of f which is the sum of all the terms of f whose monomials have weighted degree d. d must be greater than or equal to 0. If f has no terms of weighted degree d, then the result is 0.

```
MonomialsOfDegree(P, d)
```

Given a polynomial ring P and a non-negative integer d, return an indexed set consisting of all monomials in P with total degree d. If P is graded, the grading is ignored.

```
MonomialsOfWeightedDegree(P, d)
```

Given a graded polynomial ring P and a non-negative integer d, return an indexed set consisting of all monomials in P with weighted degree d. If P has the trivial grading, then this function is equivalent to the function `MonomialsOfDegree`.

Example H105E2

We create a simple graded polynomial ring and perform various simple operations on it.

```
> P<x, y, z> := PolynomialRing(RationalField(), [1, 2, 4]);
> P;
Graded Polynomial ring of rank 3 over Rational Field
Order: Grevlex with weights [1, 2, 4]
Variables: x, y, z
Variable weights: [1, 2, 4]
> VariableWeights(P);
[ 1, 2, 4 ]
> Degree(x);
1
> Degree(y);
2
> Degree(z);
4
> Degree(x^2*y*z^3); // Weighted total degree
```

```
16
> TotalDegree(x^2*y*z^3); // Natural total degree
6
> IsHomogeneous(x);
true
> IsHomogeneous(x + y);
false
> IsHomogeneous(x^2 + y);
true
> I := ideal<P | x^2*y + z, (x^4 + z)^2, y^2 + z>;
> IsHomogeneous(I);
true
> MonomialsOfDegree(P, 4);
{@
    x^4,
    x^3*y,
    x^3*z,
    x^2*y^2,
    x^2*y*z,
    x^2*z^2,
    x*y^3,
    x*y^2*z,
    x*y*z^2,
    x*z^3,
    y^4,
    y^3*z,
    y^2*z^2,
    y*z^3,
    z^4
@}
> MonomialsOfWeightedDegree(P, 4);
{@
    x^4,
    x^2*y,
    y^2,
    z
@}
```

105.3.4 Creation of Ideals and Accessing their Bases

Within the general context of ideals of polynomial rings, the term “basis” will refer to an *ordered* sequence of polynomials which generate an ideal. (Thus a basis can contain duplicates and zero elements so is not like a basis of a vector space.)

One normally creates an ideal by the `ideal` constructor or `Ideal` function, described below. But it is also possible to create an ideal with a specific basis U and then find the coordinates of polynomials from the polynomial ring with respect to U (see the function `Coordinates` below). This is done by specifying a *fixed basis* with the `IdealWithFixedBasis` intrinsic function. In this case, when Magma computes the Gröbner basis of the ideal (see below), extra information is stored so that polynomials of the ideal can be rewritten in terms of the original fixed basis. However, the use of this feature makes the Gröbner basis computation much more expensive so an ideal should usually **not** be created with a fixed basis.

`ideal< P | L >`

Given a multivariate polynomial ring P, return the ideal of P generated by the elements of P specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of P;

(b) A set or sequence of elements of P;

(c) An ideal of P;

(d) A set or sequence of ideals of P.

`Ideal(B)`

Given a set or sequence B of polynomials from a polynomial ring P, return the ideal of P generated by the elements of B with the given basis B. This is equivalent to the above `ideal` constructor, but is more convenient when one simply has a set or sequence of polynomials.

`Ideal(f)`

Given a polynomial f from a polynomial ring P, return the principal ideal of P generated by f.

`IdealWithFixedBasis(B)`

Given a sequence B of polynomials from a polynomial ring P, return the ideal of P generated by the elements of B with the given fixed basis B. When the function `Coordinates` is called, its result will be with respect to the entries of B instead of the Gröbner basis of I. **WARNING:** this function should *only* be used when it is desired to express polynomials of the ideal in terms of the elements of B, as the computation of the Gröbner basis in this case is *very* expensive, so it should be avoided if these expressions are not wanted.

```
Basis(I)
```

Given an ideal I, return the current basis of I. If I has a fixed basis, that is returned; otherwise the current basis of I (whether it has been converted to a Gröbner basis or not – see below) is returned.

```
BasisElement(I, i)
```

Given an ideal I together with an integer i, return the i-th element of the current basis of I. This the same as `Basis(I)[i]`.

105.4 Gröbner Bases

Computation in ideals of multivariate polynomial rings is possible because of the construction of Gröbner bases of such ideals. In Magma, it is possible to create ideals and compute their Gröbner bases for polynomial rings defined not only over fields but also over general Euclidean rings.

Different monomial orderings give different Gröbner bases for a fixed ideal. When an ideal I is created from a polynomial ring P or another ideal J, then the monomial order of I is taken to be the monomial order of P or J. Ideals can only be compatible if they have the same monomial order.

105.4.1 Gröbner Bases over Fields

Gröbner bases of ideals defined over fields have been studied for some time now, and there is a large literature concerning them.

For ideals defined over fields, a basis is called *minimal* if each polynomial in it is monic and not contained in the ideal generated by all the other polynomials [CLO96, Chap. 2, §7, Def. 4]. A basis is called *reduced* if each polynomial in it is monic and, for every monomial of each polynomial in the basis, that monomial is not divisible by the leading monomial of any other polynomial in the basis (equivalently, each leading monomial does not divide any monomial in any of the other polynomials) [CLO96, Chap. 2, §7, Def. 5].

For a given fixed monomial ordering, every ideal of a polynomial ring over a field possesses a **unique** sorted minimal reduced Gröbner basis (GB) [CLO96, Chap. 2, §7, Prop. 7]. This unique Gröbner basis (with respect to the order defined by the user) will be computed automatically when needed by Magma. Before this happens, an ideal will usually possess a basis which is not a Gröbner basis, but that will be changed into the unique Gröbner basis when needed. Thus the original basis will be discarded. See the procedure `Groebner` below for details on the algorithms available.

105.4.2 Gröbner Bases over Euclidean Rings

Since V2.8 (July 2001), Magma provides facilities for computing with Gröbner bases of ideals of polynomial rings over Euclidean rings (including the important case of the integer ring $\mathbf{Z}$). Such Gröbner bases are computed in Magma by an extension, due to Allan Steel (unpublished), of Jean-Charles Faugère's F_4 algorithm [Fau99], which uses sparse linear algebra.

The current Euclidean rings in Magma supported are: the integer ring $\mathbf{Z}$, the integer residue class rings $\mathbf{Z}_m$, the univariate polynomial rings $K[x]$ over any field K, Galois rings, p-adic quotient rings, and valuation rings.

We first outline some of the things which are peculiar to Gröbner bases defined over a Euclidean ring. Let I be an ideal of a polynomial ring defined over a Euclidean ring R. A subset G of I is called a *Gröbner basis* for I in Magma if, for every $f \in I$, there exists a $g \in G$ such that the leading **term** of g divides the leading **term** of f. Recall that "leading term" here means the leading coefficient times the leading monomial, so the leading coefficient of g must divide the leading coefficient of f in the base ring R. If R were a field, then obviously the leading coefficients would be insignificant and the Gröbner basis elements could be normalized (made monic) to yield an equivalent Gröbner basis. But if R is not a field, the leading coefficients are quite significant. For example, over the ring $\mathbf{Z}$, the set $\{x^2, 2x\}$ is a Gröbner basis and the polynomial x^2 is not redundant since 2 does not divide 1, but over $\mathbf{Q}$, the polynomial x^2 would be redundant.

Note that the definition here for a Gröbner basis in Magma is actually what some authors (e.g., [AL94, Def. 4.5.6]) call a *strong* Gröbner basis. *Weak* Gröbner bases have also been defined, but strong Gröbner bases satisfy stronger conditions, yield a simple effective normal form algorithm, provide more information about the ideal, are easier to get into a unique form, and are no more difficult to compute using the algorithm implemented in Magma. Thus Magma **always** computes a strong Gröbner basis, so the distinction between weak and strong is ignored. Magma also effectively computes a D-Gröbner basis as defined in [BW93, Def. 10.4, Table 10.1], although Magma also allows Euclidean rings which are not integral domains (i.e., which have zero divisors).

Over Euclidean rings, the definition of a minimal basis is practically the same as for fields (there must be no polynomial in the ideal generated by the others and each polynomial must be normalized), but the definition of a reduced basis is more subtle. A basis is called *reduced* if each polynomial in it is normalized and if, for every term $c \cdot s$ of every polynomial in the basis (where c is the coefficient and s is the monomial), then if some other polynomial in the basis has leading term $d \cdot t$, with t dividing s, then the Euclidean quotient of c by d must be zero (the remainder will be non-zero of course). Informally, this means that each polynomial is reduced modulo all the other polynomials, where each coefficient must be reduced modulo all other appropriate leading coefficients. As an example, suppose $f_1 = x^2 + 14xy$ and $f_2 = 5y + 9$ are in $\mathbf{Z}[x, y]$. Then $\{f_1, f_2\}$ is not reduced, since the second term of f_1 can be reduced by f_2 (y divides xy and the Euclidean quotient of 14 by 5 is 2, with remainder 4). But if we were to replace f_1 by $f_1 - 2xf_2 = x^2 + 4xy - 18x$, then $\{f_1, f_2\}$ would now be reduced.

Magma's extension of Faugère's algorithm depends on sparse linear algebra over Euclidean rings. (Note also that the advanced criteria for eliminating useless pairs in [Möl88]) are also implemented in this extension to work for general Euclidean rings as well.) Magma now contains an algorithm for computing a unique echelon form of a sparse matrix over such a ring; uniqueness is ensured because there is a unique Euclidean quotient-remainder algorithm for each Euclidean ring (and zero divisors are also handled properly). Consequently, based on this unique echelon form algorithm and some other techniques, Magma ensures that a Gröbner basis over a Euclidean ring is not only minimal (contains no redundant polynomials), but it is also **reduced**, and **unique**.

Thus every ideal of a polynomial ring over a Euclidean ring possesses a **unique** sorted minimal reduced Gröbner basis (with respect to some fixed monomial ordering), just as for ideals defined over fields. Also, as for ideals defined over fields, this unique Gröbner basis will be computed automatically when needed by Magma, and before this happens, an ideal will usually possess a basis which is not a Gröbner basis, but that will be changed into the unique Gröbner basis when needed.

The uniqueness of the Gröbner basis also ensures that the normal form of an element with respect to an ideal for a fixed monomial order is always unique. All of this holds even for Euclidean rings which have zero divisors.

See the examples below for illustrations of the points made above, and also how one can effectively compute with Gröbner bases of ideals defined over rings which are not even Euclidean.

105.4.3 Construction of Gröbner Bases

The following functions and procedures allow one to construct Gröbner bases. Note that a Gröbner basis for an ideal will be automatically generated when necessary; the `Groebner` procedure below simply allows control of the algorithms used to compute the Gröbner basis.

NOTE: Magma applies a special monomial representation and a special variant of the F_4 algorithm if the ideal I is defined over $\mathbf{F}_2$ and the polynomials ${x_i}^2 + x_i$ for all i are present in the input basis of the ideal I. So if one wishes to solve a system of equations over $\mathbf{F}_2$, then one should include these polynomials in the input basis (they can be at any place and in any order; as long as there is at least one copy of ${x_i}^2 + x_i$ present for each i). Alternatively (since V2.15), one can create a boolean polynomial ring (via the function `BooleanPolynomialRing` below) and construct the ideal within this. See also Example H105E5 below.

`Groebner(I: parameters)`

(Procedure.) Explicitly force a Gröbner basis (GB) for the ideal I to be constructed. This procedure is normally not necessary, as Magma will automatically compute the GB when needed, but it does allow one to control how the GB is constructed by various parameters.

By default, the parameters are set to default values which tend to work best for the particular kinds of inputs which are given, but there exist many inputs for which setting at least one of the parameters to a non-default value will lead to a dramatic improvement. (A general strategy for the computation of GBs is very difficult to design.)

If I is defined over a Euclidean ring, then Magma always uses the extension of the Faugère algorithm directly, and of the parameters given below, only `Homogenize` is applicable. So the rest of this description assumes that I is defined over a field.

We call a GB algorithm **direct** if it takes the initial basis of the ideal I (with no structure) and computes the unique minimal reduced GB of I with respect to some monomial order. Since V2.11 (May 2004), Magma has two direct algorithms for computing GBs over fields:

(1) The Faugère F_4 algorithm [Fau99], which works by specialized sparse linear algebra and is applicable to ideals defined over a finite field or the rational field;

(2) The Buchberger algorithm [CLO96, Chap. 2, §7] for ideals defined over any field.

Both direct algorithms use the advanced criteria for eliminating useless pairs in [Möl88]. Magma also uses two **order change** algorithms which both change the GB of an ideal with respect to one monomial order to the GB with respect to another monomial order:

(1) The FGLM algorithm [FGLM93], which works by efficient linear algebra and is only applicable if I is zero-dimensional;

(2) The Gröbner Walk algorithm [CKM97].

This parameter affects the main strategy:

`Al` MonStgElt *Default : "Default"*

The parameter `Al` may be set to one of: `"Default"`, `"Direct"`, `"FGLM"` or `"Walk"`. The value `"Direct"` specifies that Magma should compute the GB of I (with respect to the order of I) by a direct algorithm alone, so that an order-conversion algorithm is not used (the parameter `Faugere` below controls which direct algorithm is used).

The alternative strategy is to compute the GB first with respect to an "easy" order, and then to convert this to the GB with respect to the order of I. Setting `Al` to the values `"FGLM"` or `"Walk"` will cause this strategy to be used, where the order change algorithm will be the FGLM algorithm or Gröbner Walk algorithm, respectively.

If no algorithm is specified, or if `"Default"` is specified, an appropriate strategy is chosen by Magma, which is usually the FGLM method if the ideal is zero-dimensional and over a finite field or the rational field, and the Walk method otherwise.

The following parameters affect the direct algorithms:

`Faugere` BoolElt *Default :* `true`

`HomogeneousWeights` BoolElt *Default :* `true`

`Homogenize` BoolElt *Default :* `true`

`DegreeStart` RngIntElt *Default :* `true`

If the parameter `Faugere` is set to `true`, then the Faugère F_4 algorithm will be used (if the field is a finite field or the rational field); otherwise the Buchberger algorithm is used.

The current implementation of the Faugère algorithm is usually very much faster than the Buchberger algorithm and usually does not take much more memory, so that it is why it is now selected by default. However, there may be examples for which it may be more desirable to use the Buchberger algorithm (particularly to save some memory).†

Since V2.12, if the input basis is not homogeneous, then Magma first attempts to find a weight vector W with respect to which the ideal is homogeneous; if such a W is found, then the "easy" order used internally for the direct algorithm (accessed by `EasyIdeal`) is taken to be the `grevlexw` order with respect to W (see subsection 105.2.4), since the GB is likely to be smaller with respect to this order. The selection of such an order may be suppressed by setting the parameter `HomogeneousWeights` to `false`.

If no appropriate `grevlexw` order is used, then setting `Homogenization` to `true` specifies that the ideal should first be *homogenized*: a GB of the homogenization of the ideal is computed and then the homogenization variable is removed and the final basis reduced. This parameter has the default value of `true` over the rational field and `false` over all other fields, since most computations are improved by these defaults.

If the parameter `DegreeStart` is set to an integer d, then any S-polynomial pairs of degree less than d will be ignored.

The following parameters affect the Faugère F_4 algorithm:

`AllPairs`	BoolElt	*Default :* `false`
`PairsLimit`	BoolElt	*Default :* 0
`ReversePairs`	BoolElt	*Default :* `false`
`HFE`	BoolElt	*Default :* `false`
`Boolean`	BoolElt	*Default :* `false`
`Nthreads`	RngIntElt	*Default :* 1

By default, the Faugère F_4 algorithm includes all pairs of the next degree at each step (see [Fau99, Sec.2.5]), since this usually produces the best performance. However, setting the parameter `AllPairs` to `true` will cause the algorithm to include *all* pairs currently in the queue at each new step; this generally makes the matrix larger and is usually less efficient, but for some inputs (e.g., inhomogeneous ideals where there are only a small number of pairs for each degree at each step) this option may yield a significant improvement.

Alternatively, setting the parameter `PairsLimit` to a positive integer n will cause the algorithm to include at most n pairs from the queue at each step; this will usually make the matrix smaller, thus saving memory, but will often also make the running time longer. Setting also the parameter `ReversePairs` to `true` will reverse the list of pairs of the current degree from which the restricted set of pairs is taken: this may help a lot for certain types of input, since this may lead to new polynomials of lower degree being found more quickly. (If there is no pairs limit, then the value

† If you encounter an example where the Faugère algorithm is significantly slower than the Buchberger algorithm, then please mail it to us (`magma@maths.usyd.edu.au`)!

of ReversePairs is irrelevant since all pairs of the current degree are taken at each step.)

If the input basis is an HFE system over $\mathbf{F}_2$ such that the secret degree d is less than or equal to 127, then one should set the HFE parameter to true. In this case, Magma can apply various optimizations which save memory and time (only pairs of degree of most 4 are considered, as this is sufficient for systems for which $d \leq 127$).

Since V2.18, if the base ring is the finite field $\mathbf{F}_p$, where p is a prime with $2 < p < 2^{23.5}$, then a multi-threaded version of the algorithm is available if POSIX threads are enabled in the current Magma version. In this case, setting the parameter Nthreads to a positive integer n will cause the F_4 algorithm to use n threads within the linear algebra phase of each step. One can alternatively use the procedure SetNthreads to set the global number of threads to a value n so that n threads are always used by default in this algorithm (unless overridden by the Nthreads parameter).

The following parameters affect the Buchberger algorithm:

ReduceInitial	BoolElt	*Default :* true
RemoveRedundant	BoolElt	*Default :* true
ReduceByNew	BoolElt	*Default :* true

Setting ReduceInitial to true specifies that the basis of the ideal should be first reduced (see the function Reduce) before any S-polynomial pairs are considered. Setting RemoveRedundant to true specifies that redundant polynomials in the input (which reduce to zero with respect to the other polynomials) should first be removed. Setting ReduceByNew to true specifies that when a new polynomial f is inserted into the current GB being constructed, the current basis should be reduced by f (thus the basis stays close to being fully reduced throughout the algorithm).

Each of these control parameters usually have the default values of true (it depends on the coefficient ring).

The following parameters affect the Walk algorithm:

SigmaEpsilon	FldRatElt	*Default :* 1/2
TauEpsilon	FldRatElt	*Default :* $1/n$
SigmaVectors	RngIntElt	*Default :* n
TauVectors	RngIntElt	*Default :* $\lceil n/2 \rceil$

The parameters SigmaEpsilon and TauEpsilon control the factor ϵ which is used in the Walk algorithm to perturb the initial weight vector σ and the final weight vector τ respectively. The parameters SigmaVectors and TauVectors determine how many weight vectors of the initial and final orders are used to perturb the initial weight vector σ and the final weight vector τ respectively. By default, the ϵ factor and number of weight vectors for σ are determined dynamically to be "optimal", while the ϵ factor for τ is taken to be $1/n$ and the number of weight vectors for τ is taken to be $\lceil n/2 \rceil$, where n is the rank of I.

GroebnerBasis(I: *parameters*)

Given an ideal I, force the Gröbner basis of I to be computed, and then return that. The parameters are the same as those for the procedure Groebner.

See also the function GroebnerBasis(S,d) below, which creates a truncated degree-d Gröbner basis.

GroebnerBasis(S: *parameters*)

Given a set or sequence S of polynomials, return the unique Gröbner basis of the ideal generated by S as a sorted sequence. This function is useful for computing Gröbner bases without the need to construct ideals. The parameters are the same as those for the procedure Groebner.

See also the function GroebnerBasis(S,d) below, which creates a truncated degree-d Gröbner basis.

GroebnerBasisUnreduced(S: *parameters*)

Homogenize	BoolElt	*Default :* true
ReduceInitial	BoolElt	*Default :* true
ReduceByNew	BoolElt	*Default :* true

Given a set or sequence S of polynomials, return an unreduced Gröbner basis of the ideal generated by S as a sorted sequence. This function is useful for computing Gröbner bases without the need to construct ideals and when the reduction of the Gröbner basis is very expensive. The parameters behave the same as for the procedure Groebner.

GroebnerBasis(S, d: *parameters*)

Given a set or sequence S of polynomials, return the degree-d Gröbner basis of the ideal generated by S, which is the truncated Gröbner basis obtained by ignoring S-polynomial pairs whose total degree is greater than d.

If the ideal is homogeneous, then it is guaranteed that the result G_d is equal to the set of all polynomials in the full Gröbner basis of the ideal whose total degree is less than or equal to d, and thus a polynomial whose total degree is less than or equal to d is in the ideal iff its normal form with respect to the degree-d Gröbner basis G_d is zero. But if the ideal is not homogeneous, these last properties may not hold, but it may be still useful to construct the truncated basis.

The parameters are the same as those for the procedure Groebner. See the section on graded polynomial rings below for an example. See also [BW93, section 10.2], for further discussion.

105.4.4 Related Functions

The following functions and procedures perform operations related to Gröbner bases.

`HasGroebnerBasis(I)`

Given an ideal I, return whether the Gröbner basis of I can be computed. This depends on the type of base ring of I: the base ring must currently be a field or a Euclidean ring.

`EasyIdeal(I)`

Given an ideal I, return the ideal E which is mathematically equal to I but whose basis is the Gröbner basis of I with respect to an "easy" order, together with an isomorphism f from I onto E. The easy order is usually the `grevlex` order or `grevlexw` order with suitable weights, and the easy basis (the Gröbner basis of the easy ideal) of I is used extensively by Magma in many of its internal algorithms; this function allows one to access this "easy" Gröbner basis directly.

`EasyBasis(I)`

Given an ideal I, return the Gröebner basis of the easy ideal of I.

`SmallBasis(I)`

Given an ideal I, return the basis of I with shortest length which is currently known. This may be the original basis with which I was constructed, or a Gröbner basis, but the result is always has the the same monomial order as the main monomial order of I.

`MarkGroebner(I)`

(Procedure.) Given an ideal I, mark the current basis of I to be *the* Gröbner basis of the ideal w.r.t. the monomial order of the ideal. Note that the current basis must exactly equal the unique (reverse) sorted minimal reduced Gröbner basis for the ideal, as returned by the function `GroebnerBasis`. This procedure is useful when one creates an ideal with a basis known to be the Gröbner basis of the ideal from a previous computation or for other reasons. If the basis is not the unique Gröbner basis, the results are unpredictable.

`IsGroebner(S)`

`IsGroebner(S)`

Given a set or sequence S of polynomials describing a basis of an ideal, return whether the basis is itself a (not necessarily minimal or reduced) Gröbner basis of the ideal.

`Coordinates(I, f)`

Given an ideal I of a polynomial ring P, together with a polynomial f in I, and supposing that I has basis $b_1, \ldots, b_k$, return a sequence $[g_1, \ldots, g_k]$ of elements of P so that $f = g_1 * b_1 + \ldots + g_k * b_k$. If I was created by `IdealWithFixedBasis`(B), then the fixed basis B is used as the basis $b_1, \ldots, b_k$; otherwise the (unique) Gröbner basis of I is used as the basis $b_1, \ldots, b_k$. The resulting sequence is not necessarily unique.

`CoordinateMatrix(I)`

Given an ideal I such that I has a fixed basis (i.e., such that I was created via the function `IdealWithFixedBasis`), return the coordinate matrix C of I. The i-th row of C gives the coordinates of the i-th element of the Gröebner basis of I w.r.t. the fixed basis of I. The Gröebner basis of I is first computed if it has not been already.

`NormalForm(f, I)`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return the unique normal form of f with respect to (the Gröbner basis of) I. The normal form of f is zero if and only if f is in I.

`NormalForm(f, S)`

Given a polynomial f from a polynomial ring P, together with a set or sequence S of polynomials from P, return a normal form g of f with respect to S. (This is not unique in general. If the normal form of f is zero then f is in the ideal generated by S, but the converse is false in general. In fact, the normal form is unique if and only if S forms a Gröbner basis.) If S is a sequence, one may also assign a second return value C which gives the coordinates of the reduction, so that $C[i] \cdot S[i]$ is subtracted from f for each i to yield g.

`SPolynomial(f, g)`

Given elements f and g from a polynomial ring P, return the S-polynomial of f and g.

`Reduce(S)`

Given a set or sequence S of polynomials, return the sequence consisting of the reduction of S. The reduction is obtained by reducing to normal form each element of S with respect to the other elements and sorting the resulting non-zero elements left. Note that all Gröbner bases returned by Magma are automatically reduced so that this function would usually only be used just to simplify a set or sequence of polynomials which is not a Gröbner basis.

```
ReduceGroebnerBasis(S)
```

Given a set or sequence S of polynomials which is assumed to be a (not necessarily minimal or reduced) Gröbner basis for an ideal, return the sequence consisting of the reduction of S. The reduction is obtained by first removing each redundant polynomial whose leading term is a multiple of another leading term and then reducing the remaining polynomials as in the function `Reduce`. This function would usually only be used to reduce a set or sequence of polynomials which is known to be a non-reduced Gröbner basis (created in some way other than by one of Magma's internal Gröbner basis construction algorithms).

105.4.5 Gröbner Bases of Boolean Polynomial Rings

Since V2.15, a special type of polynomial ring is available: the **boolean polynomial ring** in n variables. Such a ring is a multivariate polynomial ring defined over $\mathbf{F}_2$ but such that all monomials are reduced modulo the *field relations* $x_i^2 = x_i$ for each i (so a bit vector representation can be used for monomials). Technically, the ring is thus the quotient algebra

$$\mathbf{F}_2[x_1, \ldots, x_n]/\langle x_1^2 + x_1, \ldots, x_n^2 + x_n\rangle.$$

Besides the basic creation and access functions for elements and ideals of such a ring, the main interest is to compute and examine a Gröbner basis of an ideal. Since the field relations are always present, an ideal represents a zero-dimensional system of multivariate polynomial equations over $\mathbf{F}_2$ with the solution components always lying in $\mathbf{F}_2$; these are particularly of interest for algebraic attacks on cryptosystems. Otherwise, there are not many other operations applicable to such rings and their elements.

Note that if one creates an ideal I of $\mathbf{F}_2[x_1, \ldots, x_n]$ such that the basis of I includes the *field polynomials* ($x_i^2 + x_i$ for each i), then Magma automatically uses the boolean polynomial ring representation internally, so this is basically equivalent to using the boolean polynomial ring type, except that Magma will have to move back to the original ring $\mathbf{F}_2[x_1, \ldots, x_n]$ at the end, and this may take much more time and memory. So it is preferable to use the boolean polynomial ring from the outset if one wishes to create the Gröbner basis of such an ideal and examine it (particularly if it does not collapse down to a sequence of linear polynomials).

See example H105E5 below for simple uses of boolean polynomial rings.

```
BooleanPolynomialRing(n)
```

Create the boolean polynomial ring with n variables (whose coefficients lie in $\mathbf{F}_2$). The default monomial order chosen is the lexicographical (`lex`) order.

```
BooleanPolynomialRing(n, order)
```

Create the boolean polynomial ring with n variables (whose coefficients lie in $\mathbf{F}_2$) and with the given order *order* on the monomials. Currently, *order* must be one of the following strings: `"lex"`, `"grevlex"`, `"glex"`.

```
BooleanPolynomialRing(B, Q)
```

Given a boolean polynomial ring B of rank n and a sequence Q of integers, create the boolean polynomial in B whose monomials are given by the entries of Q: each integer must be in the range $[0 \ldots 2^n-1]$ and its binary expansion gives the exponents of the monomial in order (the resulting monomials are sorted w.r.t. the monomial order of B, so may be given in any order and duplicate monomials are added).

This function is simply provided so that boolean polynomials may be stored and read back in a compact form; otherwise, one can create a boolean polynomial in the usual way from the generators of B after B is created. Note also that if one prints B, an ideal of B, or an element of B with the `Magma` print level, then this function will be used to print the elements in a compact form.

105.4.6 Verbosity

This subsection describes the verbose flags available for the Gröbner basis algorithms. There are separate verbose flags for each algorithm (`Buchberger`, etc.), but the all-encompassing verbose flag `Groebner` includes all these flags implicitly.

For each procedure provided for setting one of these flags, the value `false` is equivalent to level 0 (nothing), and `true` is equivalent to level 1 (minimal verbosity). For each `Set-` procedure, there is also a corresponding `Get-` function to return the value of the corresponding flag.

```
SetVerbose("Groebner", v)
```

(Procedure.) Change the verbose printing level for all Gröbner basis algorithms to be v. This includes all of the algorithms whose verbosity is controlled by flags subsequently listed, as well as some other minor related algorithms. Currently the legal levels are 0, 1, 2, 3, or 4. One would normally set this flag to 1 for minimal verbosity for Gröbner basis-type computations, and possibly also set one or more of the following flags to levels higher than 1 for more verbosity.

```
SetVerbose("Buchberger", v)
```

(Procedure.) Change the verbose printing level for the Buchberger algorithm to be v. Currently the legal levels are 0, 1, 2, 3, or 4. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

```
SetVerbose("Faugere", v)
```

(Procedure.) Change the verbose printing level for the Faugère algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

```
SetVerbose("FGLM", v)
```

(Procedure.) Change the verbose printing level for the FGLM order change algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

```
SetVerbose("GroebnerWalk", v)
```

(Procedure.) Change verbose printing for the Gröbner Walk order change algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

Example H105E3

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

```
> Q := RationalField();
> P<x, y, z, t, u, v> := PolynomialRing(Q, 6);
> I := ideal<P |
>     x + y + z + t + u + v,
>     x*y + y*z + z*t + t*u + u*v + v*x,
>     x*y*z + y*z*t + z*t*u + t*u*v + u*v*x + v*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*v + t*u*v*x + u*v*x*y + v*x*y*z,
>     x*y*z*t*u + y*z*t*u*v + z*t*u*v*x + t*u*v*x*y + u*v*x*y*z + v*x*y*z*t,
>     x*y*z*t*u*v - 1>;
> time B := GroebnerBasis(I);
Time: 1.140
> #B;
17
> B[17];
v^48 - 2554*v^42 - 399710*v^36 - 499722*v^30 + 499722*v^18 + 399710*v^12 +
    2554*v^6 - 1
> time Factorization(B[17]);
[
    <v - 1, 1>,
    <v + 1, 1>,
    <v^2 + 1, 1>,
    <v^2 - 4*v + 1, 1>,
    <v^2 - v + 1, 1>,
    <v^2 + v + 1, 1>,
    <v^2 + 4*v + 1, 1>,
    <v^4 - v^2 + 1, 1>,
    <v^4 - 4*v^3 + 15*v^2 - 4*v + 1, 1>,
    <v^4 + 4*v^3 + 15*v^2 + 4*v + 1, 1>,
    <v^8 + 4*v^6 - 6*v^4 + 4*v^2 + 1, 1>,
    <v^8 - 6*v^7 + 16*v^6 - 24*v^5 + 27*v^4 - 24*v^3 +
        16*v^2 - 6*v + 1, 1>,
    <v^8 + 6*v^7 + 16*v^6 + 24*v^5 + 27*v^4 + 24*v^3 +
        16*v^2 + 6*v + 1, 1>
]
Time: 0.060
```

Example H105E4

We solve the system of equations Runge-Kutta 2 from the paper “Some Examples for Solving Systems of Algebraic Equations by Calculating Groebner Bases” by Boege, Gebauer, and Kredel (J. Symbolic Computation (1986) **1**, 83–98). The coefficient field K is the rational function field $\mathbf{Q}(c2, c3)$, and the polynomial ring $K[c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43]$ has 11 variables with the lexicographical ordering on monomials. The resulting Gröbner basis contains a linear polynomial for each variable so there is exactly one solution to the system.

```
> K<c2, c3> := FunctionField(IntegerRing(), 2);
> P<c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43> := PolynomialRing(K, 11);
> I := ideal<P |
>    b1 + b2 + b3 + b4 - 1,
>    b2*c2 + b3*c3 + b4*c4 - 1/2,
>    b2*c2^2 + b3*c3^2 + b4*c4^2 - 1/3,
>    b3*a32*c2 + b4*a42*c2 + b4*a43*c3 - 1/6,
>    b2*c2^3 + b3*c3^3 + b4*c4^3 - 1/4,
>    b3*c3*a32*c2 + b4*c4*a42*c2 + b4*c4*a43*c3 - 1/8,
>    b3*a32*c2^2 + b4*a42*c2^2 + b4*a43*c3^2 - 1/12,
>    b4*a43*a32*c2 - 1/24,
>    c2 - a21,
>    c3 - a31 - a32,
>    c4 - a41 - a42 - a43>;
> time Groebner(I);
Time: 0.110
> I;
Ideal of Polynomial ring of rank 11 over Multivariate rational function field
    of rank 2 over Integer Ring
Order: Lexicographical
Variables: c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43
Inhomogeneous, Dimension 0
Groebner basis:
[
    c4 - 1,
    b4 + (-6*c2*c3 + 4*c2 + 4*c3 - 3)/(12*c2*c3 - 12*c2 - 12*c3 + 12),
    b3 + (2*c2 - 1)/(12*c2*c3^2 - 12*c2*c3 - 12*c3^3 + 12*c3^2),
    b2 + (-2*c3 + 1)/(12*c2^3 - 12*c2^2*c3 - 12*c2^2 + 12*c2*c3),
    b1 + (-6*c2*c3 + 2*c2 + 2*c3 - 1)/(12*c2*c3),
    a21 - c2,
    a31 + (-4*c2^2*c3 + 3*c2*c3 - c3^2)/(4*c2^2 - 2*c2),
    a32 + (-c2*c3 + c3^2)/(4*c2^2 - 2*c2),
    a41 + (-12*c2^2*c3^2 + 12*c2^2*c3 - 4*c2^2 + 12*c2*c3^2 - 15*c2*c3 + 6*c2 -
        4*c3^2 + 5*c3 - 2)/(12*c2^2*c3^2 - 8*c2^2*c3 - 8*c2*c3^2 + 6*c2*c3),
    a42 + (-c2^2 + 4*c2*c3^2 - 5*c2*c3 + 3*c2 - 4*c3^2 + 5*c3 - 2)/(12*c2^3*c3 -
        8*c2^3 - 12*c2^2*c3^2 + 6*c2^2 + 8*c2*c3^2 - 6*c2*c3),
    a43 + (-2*c2^2*c3 + 2*c2^2 + 3*c2*c3 - 3*c2 - c3 + 1)/(6*c2^2*c3^2 -
        4*c2^2*c3 - 6*c2*c3^3 + 3*c2*c3 + 4*c3^3 - 3*c3^2)
]
```

Example H105E5

We demonstrate how one can solve a system of multivariate equations over $\mathbf{F}_2$. We construct a sequence B of 4 polynomials in 5 variables, and note that the Gröbner basis of B contains monomials having degrees greater than 1.

```
> P<a,b,c,d,e> := PolynomialRing(GF(2), 5);
> B := [a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1];
> GroebnerBasis(B);
[
    a + c^2*d + c + d^2*e,
    b*c + d^3*e^2 + d^3*e + d^2*e^2 + d*e + e + 1,
    b*e + d*e^2 + d*e + e,
    c*e + d^3*e^2 + d^3*e + d^2*e^2 + d*e,
    d^4*e^2 + d^4*e + d^3*e + d^2*e^2 + d^2*e + d*e + e
]
```

If one wanted to consider solutions over an algebraic closure of $\mathbf{F}_2$, then one would have to work with this ideal. But to solve over $\mathbf{F}_2$ itself, one can add the *field polynomials* $a^2 + a$, $b^2 + b$, etc. Magma recognizes these extra polynomials and uses an optimized representation; this makes the computation much faster for larger examples. The resulting polynomials (besides any remaining field polynomials) will always have degree at most 1 in each variable. In this example, we see that there are 2 solutions over $\mathbf{F}_2$ for the system.

```
> L := [P.i^2 + P.i: i in [1 .. Rank(P)]];
> BB := B cat L;
> BB;
[
    a*b + c*d + 1,
    a*c*e + d*e,
    a*b*e + c*e,
    b*c + c*d*e + 1,
    a^2 + a,
    b^2 + b,
    c^2 + c,
    d^2 + d,
    e^2 + e
]
> GroebnerBasis(BB);
[
    a + d + 1,
    b + 1,
    c + 1,
    d^2 + d,
    e
]
> I := ideal<P|BB>;
> Variety(I);
[ <0, 1, 1, 1, 0>, <1, 1, 1, 0, 0> ]
```

Since V2.15, an alternative way to solve the system over $\mathbf{F}_2$ is to use the boolean polynomial ring type as follows.

```
> P<a,b,c,d,e> := BooleanPolynomialRing(5, "grevlex");
> B := [a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1];
> I := Ideal(B);
> I;
Ideal of Boolean polynomial ring of rank 5 over GF(2)
Order: Graded Reverse Lexicographical (bit vector word)
Variables: a, b, c, d, e
Basis:
[
    a*b + c*d + 1,
    a*c*e + d*e,
    a*b*e + c*e,
    c*d*e + b*c + 1
]
> GroebnerBasis(I);
[
    a + d + 1,
    b + 1,
    c + 1,
    e
]
> Variety(I);
[ <0, 1, 1, 1, 0>, <1, 1, 1, 0, 0> ]
```

In general, if one wishes to solve a system over $\mathbf{F}_2$ from the outset, it is best to use the boolean polynomial ring type so as to save memory (and to avoid internal conversion to and from the bit vector representation for monomials). Note also that because of the implicit field relations, the Gröbner basis of an ideal generated by only one polynomial may have several polynomials. In the following example, the Gröbner basis of an ideal generated by just one polynomial has linear polynomials alone.

```
> R<[x]> := BooleanPolynomialRing(10, "grevlex");
> R;
Boolean polynomial ring of rank 10 over GF(2)
Order: Graded Reverse Lexicographical (bit vector word)
Variables: x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], x[9], x[10]
> f := x[2]*x[3]*x[5]*x[7] + x[2]*x[4]*x[5]*x[8] + x[3]*x[4]*x[5]*x[9] +
>    x[3]*x[6]*x[7]*x[9] + x[2]*x[3]*x[5] + x[2]*x[4]*x[5] + x[2]*x[3]*x[7] +
>    x[2]*x[5]*x[7] + x[3]*x[5]*x[7] + x[3]*x[6]*x[7] + x[2]*x[4]*x[8] +
>    x[2]*x[5]*x[8] + x[4]*x[5]*x[8] + x[3]*x[6]*x[9] + x[3]*x[7]*x[9] +
>    x[6]*x[7]*x[9] + x[2]*x[3] + x[2]*x[4] + x[3]*x[5] + x[4]*x[5] +
>    x[3]*x[6] + x[2]*x[7] + x[5]*x[7] + x[6]*x[7] + x[2]*x[8] + x[4]*x[8] +
>    x[5]*x[8] + x[3]*x[9] + x[6]*x[9] + x[7]*x[9] + x[1]*x[10] + x[1] + x[4]
>    + x[6] + x[8] + x[9] + x[10];
> I := Ideal([f]);
> G := GroebnerBasis(I);
> #G;
38
> [Length(f): f in G];
[ 188, 50, 80, 82, 26, 22, 20, 26, 20, 20, 26, 32, 8, 8, 8, 8, 32, 32, 8, 8, 8,
8, 8, 8, 8, 8, 8, 32, 8, 8, 8, 8, 40, 5, 8, 8, 8, 8 ]
> G[38];
x[1]*x[4]*x[7]*x[10] + x[1]*x[5]*x[7]*x[10] + x[1]*x[4]*x[7] + x[1]*x[5]*x[7] +
    x[4]*x[7]*x[10] + x[5]*x[7]*x[10] + x[4]*x[7] + x[5]*x[7]
```

Example H105E6

This simple example illustrates some of the peculiarities of Gröbner bases over Euclidean rings. We first create a simple ideal I in $\mathbf{Z}[x, y, z]$ and compute its Gröbner basis.

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> I := ideal<P| x^2 - 1, y^2 - 1, 2*x*y - z>;
> GroebnerBasis(I);
[
    x^2 - 1,
    x*z - 2*y,
    2*x - y*z,
    y^2 - 1,
    z^2 - 4
]
```

Notice that the Gröbner basis contains polynomials whose leading terms are x^2, xz and $2x$, but the third cannot eliminate the first two since the leading coefficient 2 does not divide the other leading coefficients 1 and 1.

When we compute normal forms modulo I, x is clearly not reducible by any polynomial, while $2x$ can be reduced by the $2x - yz$ polynomial.

```
> NormalForm(x, I);
x
> NormalForm(2*x, I);
y*z
```

If we compute the normal form of $(-x)$ modulo I, then even though the x monomial cannot be reduced, the result is NOT the negative of the normal form of x, since one can use the $2x - yz$ polynomial and the fact that $((-1) \bmod 2)$ is 1 to reduce the polynomial to a unique normal form. This behaviour differs from that for ideals defined over fields, where the normal form of $-f$ will always be the negative of the normal form of f.

```
> NormalForm(-x, I);
```

```
x - y*z
```

If we reduce the Gröbner basis modulo various primes, we obtain familiar Gröbner bases over fields:

```
> GroebnerBasis(ChangeRing(I, GF(2)));
[
    x^2 + 1,
    y^2 + 1,
    z
]
> GroebnerBasis(ChangeRing(I, GF(3)));
[
    x + y*z,
    y^2 + 2,
    z^2 + 2
]
```

But if we reduce modulo 4, using the ring of integers modulo 4, then the Gröbner basis still has a structure not encountered when working over fields:

```
> GroebnerBasis(ChangeRing(I, IntegerRing(4)));
[
    x^2 + 3,
    x*z + 2*y,
    2*x + y*z,
    y^2 + 3,
    z^2,
    2*z
]
```

In fact, the new polynomial $2z$ has been included in this Gröbner basis.

Example H105E7

This example shows how one can use Gröbner bases over the integers to find the primes modulo which a system of equations has a solution, when the system has no solutions over the rationals.

We first form a certain ideal I in $\mathbf{Z}[x, y, z]$, and note that the Gröbner basis of I over $\mathbf{Q}$ contains 1, so there are no solutions over $\mathbf{Q}$ or an algebraic closure of it (this is not surprising as there are 4 equations in 3 unknowns).

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> I := ideal<P | x^2 - 3*y, y^3 - x*y, z^3 - x, x^4 - y*z + 1>;
> GroebnerBasis(ChangeRing(I, RationalField()));
[
    1
]
```

However, when we compute the Gröbner basis of I (defined over $\mathbf{Z}$), we note that there is a certain integer in the ideal which is not 1.

```
> GroebnerBasis(I);
```

```
[
    x + 170269749119,
    y + 2149906854,
    z + 170335012540,
    282687803443
]
```

Now for each prime p dividing this integer 282687803443, the Gröbner basis of I modulo p will be non-trivial and will thus give a solution of the original system modulo p.

```
> Factorization(282687803443);
[ <101, 1>, <103, 1>, <27173681, 1> ]
> GroebnerBasis(ChangeRing(I, GF(101)));
[
    x + 19,
    y + 48,
    z + 68
]
> GroebnerBasis(ChangeRing(I, GF(103)));
[
    x + 39,
    y + 8,
    z + 85
]
> GroebnerBasis(ChangeRing(I, GF(27173681)));
[
    x + 26637654,
    y + 3186055,
    z + 10380032
]
```

Of course, modulo any other prime the Gröbner basis is trivial so there are no other solutions. For example:

```
> GroebnerBasis(ChangeRing(I, GF(3)));
[
    1
]
```

Note that the problem can also be solved by using resultants, but this may yield many extraneous potential primes, while the Gröbner basis technique yields the exact list of primes for which there are modular solutions.

Example H105E8

This example shows how one can effectively compute in Magma with Gröbner bases over a ring which is not Euclidean (and may not even be a principal ideal ring), by starting with **Z** and adding appropriate defining relations. The input for this example is based on [AL94, Ex. 4.2.13].

Let $R = \mathbf{Z}[\sqrt{-5}]$. R is the maximal order of $\mathbf{Q}(\sqrt{-5})$ and is **NOT** a PIR. We consider the ideal I of $R[x, y]$ generated by $f_1 = 2xy + \sqrt{-5}y$ and $f_2 = (1 + \sqrt{-5})x^2 - xy$. To work over R, we simply compute over **Z**, introduce a new variable S to represent $\sqrt{-5}$, make sure that S is less than both x and y in the monomial order, and include the polynomial $(S^2 + 5)$ in the ideal I. We then print out the Gröbner basis of I.

```
> P<x, y, S> := PolynomialRing(IntegerRing(), 3);
> f1 := 2*x*y + S*y;
> f2 := (1 + S)*x^2 - x*y;
> I := ideal<P | f1, f2, S^2 + 5>;
> GroebnerBasis(I);
[
    x^2*S + x^2 + 5*y^3 + 13*y*S - 25*y,
    6*x^2 + 5*y^2 + 3*y*S - 10*y,
    x*y + 5*y^3 + 13*y*S - 25*y,
    y^2*S + 5*y^2 - 15*y,
    10*y^2 + 5*y*S - 25*y,
    S^2 + 5
]
```

In [AL94, p. 224], a (weak) Gröbner basis for the ideal is given as $\{f_2, f_5, f_7, f_9\}$, where $f_5 = (5 + \sqrt{-5})y^2 - 15y$, $f_7 = -2\sqrt{-5}y^2 + 5(1 + \sqrt{-5})y$, and $f_9 = xy + \sqrt{-5}y^3 - 5\sqrt{-5}y^2 + 8\sqrt{-5}y$. We can easily verify that the ideal J generated by these 4 polynomials describes the same ideal as I (and so has the same Gröbner basis in Magma).

```
> f5 := (5 + S)*y^2 - 15*y;
> f7 := -2*S*y^2 + (5 + 5*S)*y;
> f9 := x*y + S*y^3 - 5*S*y^2 + 8*S*y;
> J := ideal<P |  f2, f5, f7, f9, S^2 + 5>;
> I eq J;
true
> GroebnerBasis(I) eq GroebnerBasis(J);
true
```

We can even write f_5, f_7 and f_9 as combinations of the Gröbner basis elements of I, as follows.

```
> Coordinates(I, f5);
[
    0, 0, 0, 1, 0, 0
]
> Coordinates(I, f7);
[
    0, 0, 0, -2, 1, 0
]
> Coordinates(I, f9);
[
    0, 0, 1, y, -y - 1, 0
]
```

We can see that these elements are fairly trivially derived from the Gröbner basis which Magma computes for I. But if we now create J again using the `IdealWithFixedBasis` function and the sequence $Q = [f_2, f_5, f_7, f_9, S^2 + 5]$, then we can see the coordinates of any element of $I = J$ as a linear combination of the elements of Q. We find the coordinates of the second element of MAGMA's original Gröbner basis of I with respect to Q. The resulting coordinates are rather non-trivial.

```
> Q := [f2, f5, f7, f9, S^2 + 5];
> J := IdealWithFixedBasis(Q);
> J eq I;
true
> g := GroebnerBasis(I)[2];
> g;
6*x^2 + 5*y^2 + 3*y*S - 10*y
> C := Coordinates(J, g);
> C;
[
    -S + 1,
    -5*y + 1,
    -x - y^2*S + 7*y*S - 2*y - 7*S - 2,
    -2*y*S + 4*S + 6,
    x^2 + 5*y^3 - 13*y^2 + 3*y
]
```

We check that multiplying out the expression recovers g.

```
> &+[C[i]*Q[i]: i in [1 .. #C]] eq g;
true
```

Note that in the terminology of Adams and Loustaunau, MAGMA is here computing a "strong" Gröbner basis (for this representation which uses an extra variable for $\sqrt{-5}$), while these authors show that $\{f_2, f_5, f_7, f_9\}$ constitutes a "weak" Gröbner basis for I over the ring $\mathbf{Z}[\sqrt{-5}]$. The fact that the coordinates of g with respect to Q are rather non-trivial shows that MAGMA's strong Gröbner basis computation has computed a lot more information than the weak Gröbner basis (i.e., g, which must be included in the strong Gröbner basis, is not trivially derived from Q).

Most importantly of all, the fact that we have done all this by defining things over $\mathbf{Z}$ with the extra variable S has been no less powerful: we can still do full membership testing, normal forms, coordinate computations, etc. with this representation. Also, see below for an elimination computation which continues this example.

Gröbner bases over very many other general rings can be effectively handled in just the same way as that presented in this example! For example, if we need $\alpha = (1 + \sqrt{5})/2$, we can introduce a variable new A and the polynomial $(2A - 1)^2 - 5$.

Example H105E9

We construct an ideal I of the polynomial ring $P = \mathbf{Q}[x, y]$ with a specific fixed basis S, determine that I is the full polynomial ring P, and then find coordinates of the polynomial 1 of P with respect to S. Note that we use the function `IdealWithFixedBasis` to construct the ideal so that the fixed basis will be remembered.

```
> P<x, y> := PolynomialRing(RationalField(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
true
> C := Coordinates(I, P!1);
> C;
[
    -1/2*x^2*y^3 - 1/2*x^2*y^2 + 1/2*x^2*y + 1/2*x^2 + 1/2*x*y^3 +
        1/2*x*y^2 - 1/2*x*y - 1/2*y^4 - 1/2*y^3 + 1/2*y^2 + 1/2*y,
    1/2*x*y^3 + 1/2*x*y^2 - 1/2*x*y - 1/2*x - 1/2*y^3 - 1/2*y^2 + 1/2*y,
    -1/2*y^2 + 1
]
```

Now we check that multiplying out by the coordinates gives 1.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
1
```

Now we move the problem to being over the integer ring $\mathbf{Z}$.

```
> P<x, y> := PolynomialRing(IntegerRing(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
false
> GroebnerBasis(I);
[
    x + 1,
    y + 1,
    2
]
```

We note that 1 is not in the ideal this time, but 2 is! So we compute the coordinates of 2 with respect to I this time.

```
> C := Coordinates(I, P!2);
> C;
[
    x^2*y^2 - x^2*y - x^2 - x*y^2 + x*y + x + y^4 + y^3 - y^2 - y - 1,
    -x*y^2 + x*y + x + y^3 + y^2 - y - 1,
    -x^2 - x*y + y - 2
]
```

Note that C is the same as above, except that each polynomial has been scaled by 2 to make it integral. Finally we check again that multiplying out by the coordinates gives 2.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
2
```

Incidentally, we can see from the Gröbner basis of I over $\mathbf{Z}$ that the only solution to the system of equations described by S is the local solution $x = y = 1$ over $\mathbf{F}_2$.

Example H105E10

Gröbner bases can be constructed over any exact Euclidean ring in Magma, not just the ring of integers and its residue class rings.

We construct an ideal I of the polynomial ring $P = \mathbf{Q}[x, y]$ with a specific fixed basis S, determine that I is the full polynomial ring P, and then find coordinates of the polynomial 1 of P with respect to S. Note that we use the function `IdealWithFixedBasis` to construct the ideal so that the fixed basis will be remembered.

```
> P<x, y> := PolynomialRing(RationalField(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
true
> C := Coordinates(I, P!1);
> C;
[
    -1/2*x^2*y^3 - 1/2*x^2*y^2 + 1/2*x^2*y + 1/2*x^2 + 1/2*x*y^3 +
        1/2*x*y^2 - 1/2*x*y - 1/2*y^4 - 1/2*y^3 + 1/2*y^2 + 1/2*y,
    1/2*x*y^3 + 1/2*x*y^2 - 1/2*x*y - 1/2*x - 1/2*y^3 - 1/2*y^2 + 1/2*y,
    -1/2*y^2 + 1
]
```

Now we check that multiplying out by the coordinates gives 1.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
1
```

Now we move the problem to being over the integer ring $\mathbf{Z}$.

```
> P<x, y> := PolynomialRing(IntegerRing(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
false
> GroebnerBasis(I);
[
    x + 1,
    y + 1,
    2
]
```

We note that 1 is not in the ideal this time, but 2 is! So we compute the coordinates of 2 with respect to I this time.

```
> C := Coordinates(I, P!2);
> C;
[
    x^2*y^2 - x^2*y - x^2 - x*y^2 + x*y + x + y^4 + y^3 - y^2 - y - 1,
    -x*y^2 + x*y + x + y^3 + y^2 - y - 1,
    -x^2 - x*y + y - 2
]
```

Note that C is the same as above, except that each polynomial has been scaled by 2 to make it integral. Finally we check again that multiplying out by the coordinates gives 2.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
2
```

Incidentally, we can see from the Gröbner basis of I over $\mathbf{Z}$ that the only solution to the system of equations described by S is the local solution $x = y = 1$ over $\mathbf{F}_2$.

105.4.7 Degree-d Gröbner Bases

`GroebnerBasis(S, d : ` *parameters*`)`

Given a set or sequence S of polynomials from a graded polynomial ring P, return the weighted degree-d Gröbner basis of the ideal generated by S, which is the truncated Gröbner basis obtained by ignoring S-polynomial pairs whose weighted degree (with respect to the grading on P) is greater than d.

If the ideal is homogeneous, then it is guaranteed that the result is equal to the set of all polynomials in the full Gröbner basis of the ideal whose weighted degree is less than or equal to d, and a polynomial whose weighted degree is less than or equal to d is in the ideal iff its normal form with respect to this truncated basis is zero. But if the ideal is not homogeneous, these last properties may not hold, but it may be still useful to construct the truncated basis.

The parameters are the same as those for the procedure `Groebner`. See also [BW93, section 10.2] for further discussion. Note that the base ring may be a field or Euclidean ring.

Example H105E11

We create a graded polynomial ring and compute the degree-d Gröbner basis of a sequence L of homogeneous polynomials for various d. Since the polynomials are homogeneous (with respect to the grading), we check that the result for each d contains the set of all polynomials in the full Gröbner basis of L having weighted degree less than or equal to d.

```
> P<a,b,c,d> := PolynomialRing(RationalField(), [4,3,2,1]);
> L := [a*b - c^2*d^3, b*c*d + c^3, c^2*d - d^5, a*d - b*c];
> [IsHomogeneous(f): f in L];
[ true, true, true, true ]
> [Degree(f): f in L];
[ 7, 6, 5, 5 ]
> G:=GroebnerBasis(L);
> G;
[
    a*b - d^7,
    a*c^3 + d^10,
    a*d - b*c,
    b^2*c - d^8,
    b*c^3 + d^9,
    b*c*d + c^3,
    b*d^5 + c^4,
    c^5 - d^10,
    c^2*d - d^5,
    c*d^7 - d^9
]
> #G;
10
> [Degree(f): f in G];
[ 7, 10, 5, 8, 9, 6, 8, 10, 5, 9 ]
> for D := 1 to 10 do
>     T := GroebnerBasis(L, D);
>     printf "D = %o, #GB = %o, contains all degree-D polynomials: %o\n",
>         D, #T, {f: f in G | Degree(f) le D} subset T;
> end for;
D = 1, #GB = 4, contains all degree-D polynomials: true
D = 2, #GB = 4, contains all degree-D polynomials: true
D = 3, #GB = 4, contains all degree-D polynomials: true
D = 4, #GB = 4, contains all degree-D polynomials: true
D = 5, #GB = 4, contains all degree-D polynomials: true
D = 6, #GB = 4, contains all degree-D polynomials: true
D = 7, #GB = 4, contains all degree-D polynomials: true
D = 8, #GB = 6, contains all degree-D polynomials: true
D = 9, #GB = 8, contains all degree-D polynomials: true
D = 10, #GB = 10, contains all degree-D polynomials: true
> GroebnerBasis(L, 5);
[
    a*b - d^7,
    a*d - b*c,
    b*c*d + c^3,
    c^2*d - d^5
]
> GroebnerBasis(L, 8);
[
    a*b - d^7,
    a*d - b*c,
    b^2*c - d^8,
    b*c*d + c^3,
    b*d^5 + c^4,
    c^2*d - d^5
]
```

105.5 Changing Coefficient Ring

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

`ChangeRing(I, S)`

Given an ideal I of a polynomial ring $P = R[x_1, \ldots, x_n]$ of rank n with coefficient ring R, together with a ring S, construct the ideal J of the polynomial ring $Q = S[x_1, \ldots, x_n]$ obtained by coercing the coefficients of the elements of the basis of I into S. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S. If R and S are fields and R is known to be a subfield of S and the current basis of I is a Gröbner basis, then the basis of J is marked automatically to be a Gröbner basis of J.

Example H105E12

It is better to find the Gröbner basis of an ideal over the smallest subfield possible (e.g. **Q**), then use `ChangeRing` to create the equivalent ideal over a splitting field to find the variety.

```
> P<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<P |
>     x + y + z + t + u,
>     x*y + y*z + z*t + t*u + u*x,
>     x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
>     x*y*z*t*u - 1>;
> Groebner(I);
> K<W> := CyclotomicField(5);
> J := ChangeRing(I, K);
> V := Variety(J);
> #V;
70
```

105.6 Changing Monomial Order

Often one wishes to change the monomial order of an ideal. Magma allows one to do this by use of the `ChangeOrder` function.

`ChangeOrder(I, Q)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a polynomial ring Q of rank n (with possibly a different order to that of P), return the ideal J of Q corresponding to I and the isomorphism f from P to Q. The map f simply maps $P.i$ to $Q.i$ for each i.

The point of the function is that one can change the order on monomials of I to be that of Q. When a Gröbner basis of J is needed to be calculated, Magma uses a conversion algorithm starting from a Gröbner basis of I if possible—this usually makes order conversion much more efficient than by computing a Gröbner basis of J from scratch.

`ChangeOrder(I, ` *order*`)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a monomial order *order* (see Section 105.2), construct the polynomial ring $Q = R[x_1, \ldots, x_n]$ with order *order*, and then return the ideal J of Q corresponding to I and the isomorphism f from P to Q. See the section on monomial orders for the valid values for the argument *order*. The map f simply maps $P.i$ to $Q.i$ for each i.

`ChangeOrder(I, T)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a tuple T, construct the polynomial ring $Q = R[x_1, \ldots, x_n]$ with the monomial order given by the tuple T on the monomials, and then return the ideal J of Q corresponding to I and the isomorphism f from P to Q. T must be a tuple whose components match the valid arguments for the monomial orders in Section 105.2 (or a tuple returned by the function `MonomialOrder`).

Example H105E13

We write a function `univgen` which, given a zero-dimensional ideal defined over a field, computes the univariate elimination ideal generator for a particular variable by changing order to the appropriate univariate order. Note that this function is the same as (and is in fact implemented in exactly the same way as) the intrinsic function `UnivariateEliminationIdealGenerator`. We then find the appropriate univariate polynomials for a particular ideal.

```
> function univgen(I, i)
>    // Make sure I has a Groebner basis so that
>    // the conversion algorithm will be used when
>    // constructing a Groebner basis of J
>    Groebner(I);
>    J := ChangeOrder(I, "univ", i);
>    Groebner(J);
>    return rep{f: f in Basis(J) | IsUnivariate(f, i)};
> end function;
>
> P<x, y, z> := PolynomialRing(RationalField(), 3, "grevlex");
> I := ideal<P |
>     1 - x + x*y^2 - x*z^2,
>     1 - y + y*x^2 + y*z^2,
>     1 - z - z*x^2 + z*y^2 >;
>
> univgen(I, 1);
x^21 - x^20 - 2*x^19 + 4*x^18 - 5/2*x^17 - 5/2*x^16 + 4*x^15 -
    15/2*x^14 + 129/16*x^13 + 11/16*x^12 - 103/8*x^11 +
    131/8*x^10 - 49/16*x^9 - 171/16*x^8 + 12*x^7 - 3*x^6 -
    29/8*x^5 + 15/4*x^4 - 17/16*x^3 - 5/16*x^2 + 5/16*x - 1/16
```

```
> univgen(I, 2);
y^14 - y^13 - 13/2*y^12 + 8*y^11 + 53/4*y^10 - 97/4*y^9 -
    45/8*y^8 + 33*y^7 - 25/2*y^6 - 18*y^5 + 107/8*y^4 + 5/8*y^3 -
    27/8*y^2 + 9/8*y - 1/8
> univgen(I, 3);
z^21 - z^20 - 2*z^19 + 4*z^18 - 5/2*z^17 - 5/2*z^16 + 4*z^15 -
    15/2*z^14 + 129/16*z^13 + 11/16*z^12 - 103/8*z^11 +
    131/8*z^10 - 49/16*z^9 - 171/16*z^8 + 12*z^7 - 3*z^6 -
    29/8*z^5 + 15/4*z^4 - 17/16*z^3 - 5/16*z^2 + 5/16*z - 1/16
```

105.7 Hilbert-driven Gröbner Basis Construction

Magma incorporates an implementation of the *Hilbert-driven Buchberger Algorithm* [Tra96]. This algorithm constructs the Gröbner basis of an homogeneous ideal I whose Hilbert series is known. The algorithm is often much more efficient than the conventional Buchberger algorithm since knowledge of the Hilbert series eliminates many unnecessary reductions of S-polynomials. The algorithm can also be used as an alternative to the Gröbner Walk algorithm for changing order since one can compute the Hilbert series of the ideal with respect to an easy monomial order, and then start again with the Hilbert-driven algorithm to compute the Gröbner basis with respect to the desired final order. Furthermore, the algorithm can sometimes be used to test whether an ideal has a particular Hilbert series and abort early if this is proven to be false. The algorithm is also used extensively internally in the Invariant Theory algorithms of Magma.

```
HilbertGroebnerBasis(S, H)
HilbertGroebnerBasis(S, N)
```

Let S be a set or sequence of homogeneous polynomials from the multivariate polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field, and let I be the ideal of P generated by S. Let either H be the Hilbert series $H_{P/I}(t)$ of I (as a rational function in $\mathbf{Z}(t)$) or let $N \in \mathbf{Z}[t]$ be a univariate integer polynomial such that the weighted numerator of the Hilbert series of I is N. This function attempts to construct the (reduced) Gröbner basis of I using the given Hilbert series. The weighted numerator of the Hilbert series of I is the Hilbert series $H_{P/I}(t)$ of I, multiplied by the denominator $\prod_{i=1}^{n} 1 - t^{d_i}$, where d_i is the weighted degree of the i-th variable x_i (this denominator is thus $(1-t)^n$ if P has the default grading).

If the function returns `false`, then H (or N) cannot be the correct Hilbert series (or weighted numerator of the Hilbert series) of I. Otherwise, the function returns `true` and a sequence B of polynomials which generates the same ideal as S; if H or N is correct, B will be the (reduced) Gröbner basis of I.

In more detail, let f_H be the power series corresponding to the true Hilbert series of I and let f_N be the power series corresponding to $N/(\prod_{i=1}^{n} 1 - t^{d_i})$. If $f_H = f_N$, then the function returns `true` and the correct (reduced) Gröbner basis of I. Otherwise, consider the first term at which f_N and f_H differ: if the coefficient of f_N is greater than that of f_H, then the function returns `false` (since it will not be able to construct the extra Gröbner basis polynomials needed), otherwise the function will return `true` with a partial Gröbner basis (since it concludes that it has enough Gröbner basis polynomials when it hasn't). Consequently, the algorithm is usually used when the correct Hilbert series or weighted numerator of the Hilbert series is known, or when there is a weighted numerator which is known to be greater than or equal to the correct weighted numerator of the Hilbert series.

```
SetVerbose("HilbertGroebner", v)
```

Change verbose printing for the Hilbert-driven Buchberger algorithm to be v. Currently the legal values for v are `true`, `false`, 0, or 1.

Example H105E14

We illustrate a subalgorithm of the Invariant Theory module of Magma which uses the Hilbert-driven Buchberger Algorithm.
Let R be the invariant ring of the (permutation) cyclic group G of order 4 over the field $K = \mathbf{F}_2$. Suppose we have a sequence L of 4 homogeneous invariants of degrees $1, 2, 2$, and 4 respectively. We wish to determine efficiently whether the polynomials of L constitute primary invariants for R. To check this, the ideal generated by L must be zero-dimensional and the elements of L must be algebraically independent. This is equivalent to the condition that the weighted numerator of the Hilbert series of the ideal is the product $(1 - t)(1 - t^2)^2(1 - t^4)$. If that is not the correct weighted numerator, it will be less than the correct weighted numerator so the algorithm will return whether the polynomials L do constitute primary invariants for R.

```
> K := GF(2);
> P<a,b,c,d> := PolynomialRing(K, 4);
> L := [
>     a + b + c + d,
>     a*b + a*d + b*c + c*d,
>     a*c + b*d,
>     a*b*c*d
> ];
> // Form potential Hilbert series weighted numerator
> T<t> := PolynomialRing(IntegerRing());
> N := &*[1 - t^Degree(f): f in L];
> N;
t^9 - t^8 - 2*t^7 + 2*t^6 + 2*t^3 - 2*t^2 - t + 1
> time l, B := HilbertGroebnerBasis(L, N);
Time: 0.000
> l;
true
> // Examine Groebner basis B of L:
> B;
[
    a + b + c + d,
    b^2 + d^2,
    b*c + b*d + c^2 + c*d,
    c^3 + c^2*d + c*d^2 + d^3,
    d^4
]
```

105.8 SAT solver

Magma V2.16 contains an interface to the *MiniSat* satisfiability (SAT) solver. Such a solver is given a system of boolean expressions in conjunctive normal form and determines whether there is an assignment in the variables such that all the expressions are satisfied. Magma supplies a function by which one may transform a system of boolean polynomial equations into an equivalent boolean system, and solve this via the SAT solver.

To use the interface function, the *MiniSat* program must currently be installed as a command external to Magma. At the time of writing (November 2009), the latest version of *MiniSat* can be installed as follows on most Unix/Linux systems:

(1) Download `http://minisat.se/downloads/minisat2-070721.zip` from the *MiniSat* website (`minisat.se`).

(2) Use the command `unzip minisat2-070721.zip` or equivalent to unzip the files.

(3) Change directory into `minisat/core` and run `make` there.

(4) Copy the produced executable `minisat` into a place which is in the current path when Magma is run.

`SAT(B)`

`Exclude`	[RngMPolElt]	*Default :* []
`Verbose`	BoolElt	*Default :* `true`

Given a sequence B of boolean polynomials in a rank-n boolean polynomial ring (or a rank-n polynomial ring over $\mathbf{F}_2$), call *MiniSat* on the associated boolean system and return whether the system is satisfiable, and if so, return also a solution S as a length-n sequence of elements of $\mathbf{F}_2$. (This assumes that *MiniSat* is in the executable path of external commands; see above for instructions for installing *MiniSat*).

The parameter `Exclude` may be set to a sequence $[e_1, \dots e_k]$, where each e_i is a sequence of n elements of $\mathbf{F}_2$, specifying that the potential solutions in e_i are to be excluded (this is done by adding new relations to the system to exclude the e_i). The verbose information printed by *MiniSat* may be controlled by the parameter `Verbose`.

Example H105E15

In Example H105E5, we solved a boolean polynomial system via the standard Gröbner basis method (which the function `Variety` uses). Here we solve the same system via the SAT solver. Each time we obtain a solution, we can call the function again, but excluding the solution(s) already found. We can thus find all the solutions to the system. Of course, this is not worth doing when there are large numbers of solutions, but it may be of interest to find all solutions when it is expected there is a small number of solutions.

```
> P<a,b,c,d,e> := BooleanPolynomialRing(5, "grevlex");
> B := [a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1];
> l, S := SAT(B);
> l;
true
> S;
[ 1, 1, 1, 0, 0 ]
> Universe(S);
Finite field of size 2
> [Evaluate(f, S): f in B];
[ 0, 0, 0, 0 ]
> l, S2 := SAT(B: Exclude := [S]);
> l;
true
> S2;
[ 0, 1, 1, 1, 0 ]
> [Evaluate(f, S2): f in B];
[ 0, 0, 0, 0 ]
> l, S3 := SAT(B: Exclude := [S, S2]);
> l;
false
```

105.9 Bibliography

[**AL94**] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases*, volume 3 of *Graduate studies in mathematics.* American Mathematical Society, Providence, R.I., 1994.

[**Buc65**] Bruno Buchburger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, University of Innsbruck, Austria, 1965.

[**BW93**] Thomas Becker and Volker Weispfenning. *Gröbner Bases.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1993.

[**CKM97**] Stephane Collart, Michael Kalkbrener, and Daniel Mall. Converting Bases with the Gröbner Walk. *J. Symbolic Comp.*, 24(3):465–469, 1997.

[**CLO96**] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms.* Undergraduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2nd edition, 1996.

[**CLO98**] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.

[**Fau99**] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139 (1-3):61–88, 1999.

[**FGLM93**] Jean-Charles Faugère, Patrizia Gianni, Daniel Lazard, and Teo Mora. Efficient computations of zero-dimensional Gröbner bases by change of ordering. *J. Symbolic Comp.*, 16:329–344, 1993.

[**Möl88**] H.M. Möller. On the construction of Gröbner bases using syzygies. *J. Symbolic Comp.*, 6:345–359, 1988.

[**Ste04**] Allan Steel. Gröbner Basis Timings Page.
URL:http://magma.maths.usyd.edu.au/users/allan/gb/, 2004.

[**Tra96**] Carlo Traverso. Hilbert Functions and the Buchberger Algorithm. *J. Symbolic Comp.*, 22(4):355–376, 1996.

106 POLYNOMIAL RING IDEAL OPERATIONS

Chapter 106
POLYNOMIAL RING IDEAL OPERATIONS

106.1 Introduction

This chapter describes the Magma functionality for ideals over polynomial rings. For the basics on multivariate polynomial rings and their elements, see Chapter 24. Most of the significant operations with ideals construct or utilise a previously-constructed Gröbner basis. The monomial ordering used for this basis can greatly affect the speed and memory usage of these operations. This ordering is attached to the polynomial ring in which the ideals are created. For information on Gröbner bases and the creation of polynomial rings with specified orders, see Chapter 105. That chapter also tells the user how to compute and return a Gröbner basis, or just to compute it internally for later use in the operations described below, with many additional configuration parameters to optimise the computation. Users may ignore the issue when creating the ambient polynomial rings by allowing Magma to make default choices. It is, however, highly recommended that users who wish to work with complicated ideals thoroughly acquaint themselves with the options available. Magma has an extremely powerful Gröbner basis engine and often makes sophisticated choices internally of alternative monomial orders for particular computations. Ultimately, however, the user may significantly speed up his work by a judicious choice of order. We note here that the default order is the lexicographical one, a total elimination order well suited to finding solutions of zero-dimensional systems of polynomial equations but tending to produce very large bases that can take much time and memory to compute. For homogeneous ideals of rings with the standard weighting (all variables have weight one), the grevlex order is usually the best in practise and there is theoretical justification for this. In the case that the ring has a different weighting and the ideal is homogeneous with respect to that, the weighted grevlex order is the best choice. In any case, the `EasyIdeal` and `EasyBasis` intrinsics of the Gröbner basis chapter return to the user a basis for an internally chosen good order and these "easy" bases are used in many internal functions if a basis with respect to the polynomial ring order has not already been computed and stored.

The functions and operations described here cover a wide range of commutative algebra functionality. This includes sums and intersections, colon ideals and saturations, elimination, radicals and primary decompositions, Noether normalisations and computation of Hilbert polynomials and Hilbert series.

Related chapters including other polynomial ring functionality relying on Gröbner bases are the chapter on invariant rings of finite group actions, Chapter 110, and the chapters on affine algebras (Chapter 108) and on modules over affine algebras (Chapter 109). The chapter on algebraically closed fields (Chapter 40) describes functions that allows one to compute the variety of an ideal over the algebraic closure of the base field. And, of course, the Algebraic Geometry component of Magma and parts of the Arithmetic Geometry are built upon the commutative algebra here.

106.2 Creation of Polynomial Rings and their Ideals

As noted in the introduction, for the basics on multivariate polynomial rings and their elements, including their creation, the user should refer to Chapter 24. For creation of polynomial rings with non-default (currently lexicographic) monomial ordering, the user should refer to Chapter 105. Similarly, the basic creation functions for ideals and additional basis options are described in Chapter 105. The commonest creation methods are the `ideal` constructor and the `Ideal` function.

106.3 First Operations on Ideals

In the following, note that since ideals of a full polynomial ring P are regarded as subrings of P, the ring P itself is a valid ideal as well (the ideal containing 1).

106.3.1 Simple Ideal Constructions

The following basic constructions involve no Gröbner basis computation.

`I + J`

Given ideals I and J of the same polynomial ring P, return the sum of I and J, which is the ideal generated by the generators of I and those of J.

`I * J`

Given ideals I and J of the same polynomial ring P, return the product of I and J, which is the ideal generated by the products of the generators of I and those of J.

`I ^ k`

Given an ideal I of the polynomial ring P, and an integer k, return the k-th power of I.

`I / J`

Given an ideal I of a polynomial ring P over a field and an ideal J of P, such that $J \subset I$, return the affine algebra I/J.

106.3.2 Basic Commutative Algebra Operations

The following important basic operations on ideals involve Gröbner basis computation and use the standard algorithms as described in Chapter 1.8 of [GP02], for example, unless otherwise stated.

`QuotientDimension(I)`

Given an ideal I of a polynomial ring P over a field K, return the dimension of P/I as a K-vector space. Note that this is quite different from the function `Dimension` below (which returns the Krull dimension of an ideal). If I is not of Krull dimension 0 then the vector space is infinite and `Infinity` is returned.

`ColonIdeal(I, J)`

`IdealQuotient(I, J)`

Given ideals I and J of the same polynomial ring P, return the colon ideal $I : J$ (or ideal quotient of I by J), consisting of the polynomials f of P such that $f * g$ is in I for all g in J.

`ColonIdeal(I, f)`

`IdealQuotient(I, f)`

Given an ideal I and an element f of a polynomial ring P, return the saturation (colon) ideal $I : f^\infty$, consisting of the polynomials g of P such that there exists an $i \geq 1$ with $f^i * g \in I$. An integer s with $s \geq 1$ is also returned such that $I : f^\infty = I : f^s$. Note that if s is not needed, only one return value of the function should be expected which increases the efficiency enormously. Note also that this function is *not* equivalent to taking the ideal quotient of I by the ideal of P generated by f. It is in some ways a more natural operation mathematically, corresponding to taking the full inverse image of the localised ideal I_f under the localisation map $P \rightarrow P_f$, and can be faster than the $I : f$ computation, if s is not required. In this case, the computation goes by the elimination of extra variable t from the ideal $< I, 1 - f * t >$.

`ColonIdealEquivalent(I, f)`

`Saturation(I, f)`

Given an ideal I and an element f of a polynomial ring P, return the saturation (colon) ideal $C = I : f^\infty$, and a polynomial $g \in P$ such that $C = I : \langle g \rangle$ and g is of minimal degree. The irreducible factors of g will be a subset of the irreducible factors of f (and the corresponding multiplicities may be greater or lesser, depending on how often an irreducible factor divides the ideal I).

`Saturation(I, J)`

Given ideals I and J of some polynomial ring P, return the saturation $(I : J^\infty)$: that is, the ideal $\{f \in P : \exists n > 0, f^n J \subseteq I\}$.

`Saturation(I)`

Given an ideal I of a polynomial ring P, return the saturation of I with respect to the irrelevant ideal of P – that is, the ideal of all elements of P having positive degree.

`Generic(I)`

Given an ideal I of a generic polynomial ring P, return P.

`LeadingMonomialIdeal(I)`

Given an ideal I, return the leading monomial ideal of I; that is, the ideal generated by all the leading monomials of I.

`I meet J`

Given ideals I and J of the same polynomial ring P, return the intersection of I and J.

`&meet S`

Given a set or sequence S of ideals of the same polynomial ring P, return the intersection of all the ideals of S.

`RegularSequence(I)`

`Homogeneous` BoolElt *Default :* `true`

Given an ideal I of a polynomial ring P over a field, computes and returns a maximal regular sequence in I. The algorithm used is that of Eisenbud and Sturmfels ([ES94]) that tries to construct a regular sequence of fairly sparse polynomials. If parameter `Homogeneous` is `true` (the default), and I is a homogeneous ideal with respect to the variable weights, then the regular sequence constructed will also consist of homogeneous polynomials.

`ReesIdeal(P, I)`

`ReesIdeal(P, J, I)`

`a` RngMPolElt *Default :* 1

`ReesIdeal(R, I)`

`a` RngMPolElt *Default :* 1

In each case P is a multivariate polynomial ring and I is an ideal of P. In the third case R is an affine quotient algebra of the form P/J. In the second case J is another ideal of P and we write R for the affine algebra P/J. In the first case, let $R = P$.

The Rees algebra $R(I)$ is the finitely-generated, graded polynomial algebra isomorphic to the algebra

$$R \oplus I \oplus I^2 \oplus I^3 \oplus \dots$$

where I gives the first graded part, I^2 the second etc. and the multiplication is the obvious one. Here I is thought of as an ideal of R, rather than P for the second and third signatures. *Proj* of this algebra represents the blow-up of the affine scheme *Spec*(R) along the closed subscheme defined by I (see Chapter 2, Section 7 of [Har77]).

The function returns the *Rees ideal*, K, such that, if R_1 is the generic polynomial ring of K, then R_1/K is an affine algebra isomorphic to $R(I)/ < a - torsion >$, where a is an element of P (or R in the third case) that gives a non-zero divisor in R and is 1 by default. In the first case, any such a remains a non-zero divisor in $R(I)$, so is redundant. However, in the second and third cases, a can be specified to be not equal to 1 by use of the `a` parameter. Geometrically, dividing out by a-torsion gives the coordinate ring of the maximal closed subscheme of the blow-up that is flat over the generic point and the codimension one points defined by the vanishing of a, if these points are regular.

106.3.3 Ideal Predicates

```
I eq J
```

Given two ideals I and J of the same polynomial ring P, return whether I and J are equal. Involves the use of a Gröbner basis for each ideal.

```
I ne J
```

Given two ideals I and J of the same polynomial ring P, return whether I and J are not equal. Involves the use of a Gröbner basis for each ideal.

```
I notsubset J
```

Given two ideals I and J in the same polynomial ring P return whether I is not contained in J. Involves the use of a Gröbner basis for J.

```
I subset J
```

Given two ideals I and J in the same polynomial ring P return whether I is contained in J. Involves the use of a Gröbner basis for J.

```
IsZero(I)
```

Given an ideal I of the polynomial ring P, return whether I is the zero ideal (contains zero alone).

```
IsProper(I)
```

Given an ideal I of the polynomial ring P, return whether I is proper; that is, whether I is strictly contained in P, or whether the Gröbner basis of I does not contain 1 alone.

```
IsHomogeneous(I)
```

Given an ideal I of the polynomial ring P, this function returns whether I is homogeneous with respect to the weights on the variables of P (i.e., whether I possesses a basis consisting of homogeneous polynomials alone). Checks whether the current basis of I consists of homogeneous polynomials and, if not and the current basis isn't Gröbner, then whether an easy Gröbner basis consists of homogeneous elements.

```
IsPrincipal(I)
```

Given an ideal I of the polynomial ring P, return whether I is principal, and if so, return also a generator of I. This will be true if and only if an arbitrary Gröbner basis consists of a single (generating) element.

```
IsPrimary(I)
```

Given an ideal I of the polynomial ring P, return whether I is primary. An ideal I is primary if and only if for all $ab \in I$, either $a \in I$ or $b^n \in I$ for some $n \geq 1$. The restrictions on I are the same as for the function `PrimaryDecomposition`—see the description of that function. In general, this function computes or retrieves the primary decomposition and checks whether it has a unique element.

`IsPrime(I)`

Given an ideal I of the polynomial ring P, return whether I is prime. An ideal I is prime if and only if for all $ab \in I$, either $a \in I$ or $b \in I$. The restrictions on I are the same as for the function `PrimaryDecomposition`—see the description of that function. Again, this function computes the primary decomposition or uses the already stored one.

`IsMaximal(I)`

Given an ideal I of the polynomial ring P, return whether I is maximal. The restrictions on I are the same as for the function `PrimaryDecomposition`—see the description of that function. Checks first whether I is zero-dimensional (see below) and, if so, then checks whether it is prime. NB: given that I is of dimension 0, the prime/primary decomposition computation is relatively fast.

`IsRadical(I)`

Given an ideal I of the polynomial ring P, return whether I is radical; that is, whether the radical of I is I itself. The restrictions on I are the same as for the function `Radical`—see the description of that function. The function computes the radical or uses the already stored one.

`IsZeroDimensional(I)`

Given an ideal I of the polynomial ring P, defined over a field, return whether I is zero-dimensional (so the quotient of P by I has non-zero finite dimension as a vector space over the coefficient field – see the section on dimension for further details). Note that the full polynomial ring P as an ideal of itself has dimension -1, so it is not zero-dimensional.

`HasGrevlexOrder(I)`

Given an ideal I of the polynomial ring P, return whether the monomial order of I is the `grevlex` order.

Example H106E1

We construct some ideals in $\mathbf{Q}[x, y, z]$ and perform basic arithmetic on them.

```
> P<x,y,z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x*y - 1, x^3*z^2 - y^2, x*z^3 - x - 1>;
> J := ideal<P | x*y - 1, x^2*z - y, x*z^3 - x - 1>;
> A := I * J;
> A;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Basis:
[
    x^2*y^2 - 2*x*y + 1,
    x^3*y*z - x^2*z - x*y^2 + y,
    x^2*y*z^3 - x^2*y - x*y - x*z^3 + x + 1,
    x^4*y*z^2 - x^3*z^2 - x*y^3 + y^2,
    x^5*z^3 - x^3*y*z^2 - x^2*y^2*z + y^3,
    x^4*z^5 - x^4*z^2 - x^3*z^2 - x*y^2*z^3 + x*y^2 + y^2,
    x^2*y*z^3 - x^2*y - x*y - x*z^3 + x + 1,
    x^3*z^4 - x^3*z - x^2*z - x*y*z^3 + x*y + y,
    x^2*z^6 - 2*x^2*z^3 + x^2 - 2*x*z^3 + 2*x + 1
]
> M := I meet J;
> M;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Basis:
[
    x^4 + x^3 - x*z^2 + z^12 - 4*z^9 + 6*z^6 - z^4 - 4*z^3 + z + 1,
    x^5 + x^4 - x^2*z^2 + z^9 - 3*z^6 + 3*z^3 - z - 1,
    x*z^3 - x - 1,
    y - z^3 + 1
]
> A eq M;
true
> QuotientDimension(A);
24
> ColonIdeal(I, J);
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0
Basis:
[
    x*y - 1,
    x^3*z^2 - y^2,
    x*z^3 - x - 1
]
```

106.3.4 Element Operations with Ideals

```
f in I
```

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is in I. The function computes the normal form of f relative to some Gröbner basis of I and checks if this is zero.

```
f notin I
```

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is not in I. As with `in`, this performs a normal form computation.

```
IsInRadical(f, I)
```

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is in the radical of I. Note that using this function is much quicker in general than actually computing the radical of I. It uses the algorithm described in section 1.8.6 of [GP02].

```
JacobianIdeal(f)
```

Return the ideal generated by all first partial derivatives of the polynomial f.

Example H106E2

We demonstrate the element operations with respect to an ideal of $\mathbf{Q}[x, y, z]$.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | (x + y)^3, (y - z)^2, y^2*z + z>;
> NormalForm(y^2*z + z, I);
0
> NormalForm(x^3, I);
-3*x^2*y - 3*x*z^4 - 6*x*z^2 + 1/2*z^3 + 3/2*z
> NormalForm(z^4 + y^2, I);
2*z^4 + 2*z^2
> x + y in I;
false
> IsInRadical(x + y, I);
true
> IsInRadical((x + y)^2, I);
true
> IsInRadical(z, I);
false
> SPolynomial(x^4 + y - z, x^2 + y - z);
-x^2*y + x^2*z + y - z
```

106.4 Computation of Varieties

The slightly non-standard term variety in this section refers to the (finite) solution set of the system of polynomials that make up a zero-dimensional ideal. It can also be thought of as the set of points of a zero-dimensional affine scheme over a specified field extension of the polynomial ring base field. For more general functionality for schemes of arbitrary dimension, see the chapters on Algebraic and Arithmetic Geometry. The functions here also work for higher-dimensional ideals if the base field is finite, when the solution set is again finite (over the base or a finite extension of the base).

The functions compute solutions over the base field of the polynomial ring or over an extension field L. Magma's algebraically closed fields (see Chapter 40) may be used to get all solutions if so desired when an explicit splitting field is not known for the system. L should be an exact field over which Magma has a root-finding algorithm for univariate polynomials or a real or complex field.

For the corresponding functions with argument a zero-dimensional scheme which may not be affine, see the Section 112.7 in the Schemes chapter.

```
Variety(I)
Variety(I, L)
```

`Digits` RngIntElt *Default :* 38

Given a zero-dimensional ideal I of a polynomial ring P, return the variety of I over its coefficient field K as a sequence of tuples. Each tuple is of length n, where n is the rank of P, and corresponds to an assignment of the n variables of P (in order) such that all polynomials in I vanish with this assignment.

If K is not a finite field then the ideal must be the full polynomial ring or be zero-dimensional so that the variety is known to be finite. If a superfield L of K is also given, the variety is computed over L instead, so the entries of the tuples lie in L.

If the field over which the variety is computed is the free complex field, Magma uses a special root finding algorithm to ensure the precision of the results; in this case, the parameter `Digits` may be given (see the `Roots` function in the Real and Complex Fields chapter (Chapter 25)).

The function works in the zero-dimensional case by first computing a triangular or radical decomposition of I (see Section 106.11). This reduces the problem to successively computing roots of univariate polynomials.

```
VarietySequence(I)
VarietySequence(I, L)
```

`Digits` RngIntElt *Default :* 38

Given a zero-dimensional ideal I of a polynomial ring P whose order is of lexicographic type, return the variety of I over its coefficient field K as a sequence of sequences of elements of K. Each inner sequence is of length n, where n is the rank of P, and corresponds to an assignment of the n variables of P (in order) such that all polynomials in I vanish with this assignment.

If K is not a finite field then the ideal must be the full polynomial ring or be zero-dimensional so that the variety is known to be finite. If a superfield L of K is also given, the variety is computed over L instead, so the entries of the sequences lie in L.

If the field over which the variety is computed is the free complex field, Magma uses a special root finding algorithm to ensure the precision of the results; in this case, the parameter `Digits` may be given (see the `Roots` intrinsic function in the Real and Complex Fields chapter (Chapter 25)).

The function works in the zero-dimensional case by first computing a triangular or radical decomposition of I (see Section 106.11). This reduces the problem to successively computing roots of univariate polynomials.

`VarietySizeOverAlgebraicClosure(I)`

Given a zero-dimensional ideal I of a polynomial ring P over a field K, return the size of the variety of I over the algebraic closure K' of K. The size is determined by finding the (prime) radical decomposition of I and placing each component of the decomposition into normal position so the size of the variety of the component over K' can be read off. Note that this function will usually be much faster than actually computing the variety of I over a suitable extension field of K.

Example H106E3

We construct an ideal I of the polynomial ring $\mathbf{F}_{27}[x, y]$, and then find the variety $V = V(I)$. We then check that I vanishes on V.

```
> K<w> := GF(27);
> P<x, y> := PolynomialRing(K, 2);
> I := ideal<P | x^8 + y + 2, y^6 + x*y^5 + x^2>;
> Groebner(I);
> I;
Ideal of Polynomial ring of rank 2 over GF(3^3)
Order: Lexicographical
Variables: x, y
Inhomogeneous, Dimension 0
Groebner basis:
[
    x + 2*y^47 + 2*y^45 + y^44 + 2*y^43 + y^41 + 2*y^39 + 2*y^38 + 2*y^37 +
        2*y^36 + y^35 + 2*y^34 + 2*y^33 + y^32 + 2*y^31 + y^30 + y^28 + y^27 +
        y^26 + y^25 + 2*y^23 + y^22 + y^21 + 2*y^19 + 2*y^18 + 2*y^16 + y^15 +
        y^13 + y^12 + 2*y^10 + y^9 + y^8 + y^7 + 2*y^6 + y^4 + y^3 + y^2 + y +
        2,
    y^48 + y^41 + 2*y^40 + y^37 + 2*y^36 + 2*y^33 + y^32 + 2*y^29 + y^28 +
        2*y^25 + y^24 + y^2 + y + 1
]
> V := Variety(I);
> V;
[ <w^14, w^12>, <w^16, w^10>, <w^22, w^4> ]
```

```
> // Check that the original polynomials vanish:
> [
>    <x^8 + y + 2, y^6 + x*y^5 + x^2> where x is v[1] where y is v[2]: v in V
> ];
[ <0, 0>, <0, 0>, <0, 0> ]
> // Note that the variety of I would be larger over an extension field of K:
> VarietySizeOverAlgebraicClosure(I);
48
```

106.5 Multiplicities

This section contains some useful invariants for an isolated singularity at the origin of a hypersurface given by a multivariate polynomial f.

```
MilnorNumber(f)
```

Given a polynomial $f \in K[x_1, \ldots, x_n]$, where K is a field, return the Milnor number of f at the origin. This is the dimension of the quotient by the ideal generated by the partials of f in the localization of $K[x_1, \ldots, x_n]$ at the origin. See [CLO98, p. 147] or [DL06, Remark 9.37].

```
TjurinaNumber(f)
```

Given a polynomial $f \in K[x_1, \ldots, x_n]$, where K is a field, return the Tjurina number of f at the origin. This is the dimension of the quotient by the ideal generated by f and the partials of f in the localization of $K[x_1, \ldots, x_n]$ at the origin. See [CLO98, p. 148] or [DL06, Def. 9.35].

Example H106E4

We compute some Milnor and Tjurina numbers, based on Exercise 12 of [CLO98, p. 177].

```
> P<x,y> := PolynomialRing(RationalField(), 2);
> MilnorNumber((x^2 + y^2)^3 - 4*x^2*y^2); // 4-leaved rose
13
> [MilnorNumber(y^2 - x^n): n in [1 .. 5]];
[ 0, 1, 2, 3, 4 ]
> P<x,y,z> := PolynomialRing(RationalField(), 3);
> [MilnorNumber(x*y*z + x^n + y^n + z^n): n in [1 .. 10]];
[ 0, 1, 8, 11, 14, 17, 20, 23, 26, 29 ]
> [TjurinaNumber(x*y*z + x^n + y^n + z^n): n in [1 .. 10]];
[ 0, 1, 8, 10, 13, 16, 19, 22, 25, 28 ]
```

A much larger example is given in [DL06, p. 254].

```
> f := y^2 - 2*x^28*y - 4*x^21*y^17 + 4*x^14*y^33 - 8*x^7*y^49 +
>      x^56 + 20*y^65 + 4*x^49*y^16;
> time TjurinaNumber(f);
2260
```

```
Time: 0.010
```

106.6 Elimination

Elimination theory plays an important role when working with ideals of multivariate polynomial rings. Magma provides an assortment of functions to perform various kinds of elimination easily. Elimination of variables is accomplished by computing a Gröbner basis with respect to a suitable elimination order (for more information about elimination orders, see Section 105.2 as well as comments in the function descriptions below).

All of the functions in this section may be applied to ideals over general Euclidean rings, not just over fields.

106.6.1 Construction of Elimination Ideals

`EliminationIdeal(I, k: parameters)`

Given an ideal I of a polynomial ring P of rank n with $P = R[x_1, \ldots, x_n]$, together with an integer k with $0 \leq k \leq n$, return the k-th elimination ideal I_k of I, which is defined to be $I \cap R[x_{k+1}, \ldots, x_n]$. Thus I_k consists of all polynomials of I which have the first k variables eliminated. If the elimination ideals I_k are to be computed for several different k, it is recommended first that a Gröbner basis with respect to lexicographical order for I first be computed as then the elimination ideals can be determined trivially. If I does not have a Gröbner basis stored with respect to lexicographical order, then a Gröbner basis computation will be necessary each time an elimination ideal is desired.

If $k = n$, then $I \cap R$ is returned, which, if R is a field, is always the full ring P or the empty ideal, according to whether I is the full polynomial ring or not. But if R is not a field, then this intersection will yield the ideal generated by the normalized smallest element of R which is in I (according to the Euclidean norm), which could be neither 0 nor 1.

The parameters are as for the `Groebner` procedure. Note that setting `Al := "Direct"` occasionally produces much better performance since the relevant elimination order may yield a better Gröbner basis than the default method of going via the `grevlex` order.

`EliminationIdeal(I, S)`

`EliminationIdeal(I, S)`

Given an ideal I of a polynomial ring P of rank n with $P = R[x_1, \ldots, x_n]$, together with a set S describing a subset U of the variables $\{x_1, \ldots x_n\}$, return the elimination ideal I_U of I, which is defined to be $I \cap R[U]$. Thus I_U consists of all polynomials of I which contain variables only found in U. U can be specified in two ways: either as a set S of integers in the range $1 \ldots n$ such the integer i corresponds to the i-th variable x_i, or as a set S of variables lying in P. S may be the empty set, in which case this is equivalent to `EliminationIdeal(I, n)`; see above.

Example H106E5

This example continues the example above which computed a Gröbner basis over a ring which was not even a PIR.

As before, $R = \mathbf{Z}[\sqrt{-5}]$ and I is the ideal of $R[x, y]$ generated by $f_1 = 2xy + \sqrt{-5}y$ and $f_2 = (1+\sqrt{-5})x^2 - xy$. As before, we compute over $\mathbf{Z}$, introduce a new variable S and include (S^2+5) in I, so we can effectively work over R.

```
> P<x, y, S> := PolynomialRing(IntegerRing(), 3);
> f1 := 2*x*y + S*y;
> f2 := (1 + S)*x^2 - x*y;
> I := ideal<P | f1, f2, S^2 + 5>;
> GroebnerBasis(I);
[
    x^2*S + x^2 + 5*y^3 + 13*y*S - 25*y,
    6*x^2 + 5*y^2 + 3*y*S - 10*y,
    x*y + 5*y^3 + 13*y*S - 25*y,
    y^2*S + 5*y^2 - 15*y,
    10*y^2 + 5*y*S - 25*y,
    S^2 + 5
]
```

In [AL94, Ex. 4.3.8], the elimination ideal $E_y = I \cap (\mathbf{Z}[\sqrt{-5})[y]$ is shown to be generated by $f_5 = (5 + \sqrt{-5})y^2 - 15y$ and $f_7 = -2\sqrt{-5}y^2 + 5(1 + \sqrt{-5})y$. We can compute E_y in Magma easily using `EliminationIdeal`. We must be careful to include S in the second argument (the set of variables which we want), since S should be considered a 'constant' (member of R) in this context.

```
> Ey := EliminationIdeal(I, {y, S});
> GroebnerBasis(Ey);
[
    y^2*S + 5*y^2 - 15*y,
    10*y^2 + 5*y*S - 25*y,
    S^2 + 5
]
```

Obviously, the polynomials yielded are simply the last 3 polynomials of the full Gröbner basis given above. We check also that the ideal generated by f_5 and f_7 over R is the same as that given by Magma.

```
> f_5 := y^2*S + 5*y^2 - 15*y;
> f_7 := -2*y^2*S + 5*y*S + 5*y;
> E := ideal<P | f_5, f_7, S^2 + 5>;
> E eq Ey;
true
```

Finally, we also compute $E_x = I \cap (\mathbf{Z}[\sqrt{-5})[x]$, which requires more effort this time, since it cannot be read off the Gröbner basis.

```
> Ex := EliminationIdeal(I, {x, S});
> GroebnerBasis(Ex);
```

```
[
    2*x^3*S + 2*x^3 + x^2*S - 5*x^2,
    12*x^3 + 6*x^2*S,
    S^2 + 5
]
```

From this, we see that E_x is generated by $(2+2\sqrt{-5})x^3 + (-5+\sqrt{-5})x^2$ and $12x^3 + 6\sqrt{-5}x^2$.

106.6.2 Univariate Elimination Ideal Generators

```
UnivariateEliminationIdealGenerator(I, i)
```

Given a zero-dimensional ideal I of a polynomial ring P of rank n with $P = K[x_1, \ldots, x_n]$, together with an integer i with $1 \leq i \leq n$, return the unique monic generator of the univariate elimination ideal $I \cap K[x_i]$.

```
UnivariateEliminationIdealGenerators(I)
```

Given a zero-dimensional ideal I of a polynomial ring P of rank n with $P = K[x_1, \ldots, x_n]$, return the sequence of length n whose i-th element is the unique monic generator of the univariate elimination ideal $I \cap K[x_i]$.

Example H106E6

We construct an ideal I (derived from Neural networks theory) of the polynomial ring $\mathbf{Q}[x, y, z]$, and then find various elimination ideals of I.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P |
>     1 - x + x*y^2 - x*z^2,
>     1 - y + y*x^2 + y*z^2,
>     1 - z - z*x^2 + z*y^2 >;
> UnivariateEliminationIdealGenerator(I, 1);
x^21 - x^20 - 2*x^19 + 4*x^18 - 5/2*x^17 - 5/2*x^16 + 4*x^15 - 15/2*x^14 +
    129/16*x^13 + 11/16*x^12 - 103/8*x^11 + 131/8*x^10 - 49/16*x^9 -
    171/16*x^8 + 12*x^7 - 3*x^6 - 29/8*x^5 + 15/4*x^4 - 17/16*x^3 - 5/16*x^2
    + 5/16*x - 1/16
> UnivariateEliminationIdealGenerator(I, 2);
y^14 - y^13 - 13/2*y^12 + 8*y^11 + 53/4*y^10 - 97/4*y^9 - 45/8*y^8 + 33*y^7 -
    25/2*y^6 - 18*y^5 + 107/8*y^4 + 5/8*y^3 - 27/8*y^2 + 9/8*y - 1/8
> E :=  EliminationIdeal(I, {y, z});
> E;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Basis:
[
    z^21 - z^20 - 2*z^19 + 4*z^18 - 5/2*z^17 - 5/2*z^16 + 4*z^15 - 15/2*z^14 +
        129/16*z^13 + 11/16*z^12 - 103/8*z^11 + 131/8*z^10 - 49/16*z^9 -
        171/16*z^8 + 12*z^7 - 3*z^6 - 29/8*z^5 + 15/4*z^4 - 17/16*z^3 - 5/16*z^2
        + 5/16*z - 1/16,
    y + 141944208/7806653*z^20 - 42803108/7806653*z^19 - 290535348/7806653*z^18
        + 309392460/7806653*z^17 - 164881460/7806653*z^16 -
        331099258/7806653*z^15 + 203830442/7806653*z^14 - 894960798/7806653*z^13
        + 622205873/7806653*z^12 + 1352184655/31226612*z^11 -
        4746138097/31226612*z^10 + 5122044359/31226612*z^9 +
        991547639/31226612*z^8 - 830598655/7806653*z^7 + 1472712995/15613306*z^6
        - 59983627/15613306*z^5 - 486698319/15613306*z^4 + 174173263/7806653*z^3
        - 30672252/7806653*z^2 - 735083/664396*z + 30093391/31226612
]
```

Example H106E7

We write a simple function ZRadical to compute the radical of a zero dimensional ideal defined over a field using the univariate elimination ideal generators. See [BW93, p. 345].

```
> function ZRadical(I)
>     // Find radical of zero dimensional ideal I
>     P := Generic(I);
>     n := Rank(P);
>     G := UnivariateEliminationIdealGenerators(I);
>     N := {};
>
>     for i := 1 to n do
>         // Set FF to square-free part of the i-th univariate
>         // elimination ideal generator
>         F := G[i];
>         FF := F;
>         while true do
>             D := GCD(FF, Derivative(FF, 1, i));
>             if D eq 1 then
>                 break;
>             end if;
>             FF := FF div D;
>         end while;
>         // Include FF in N if FF is a proper divisor of F
>         if FF ne F then
>             Include(~N, FF);
>         end if;
>     end for;
>
>     // Return the sum of I and N
>     if #N eq 0 then
>       return I;
>     else
>       return ideal<P | I, N>;
>     end if;
> end function;
```

We now apply ZRadical to an ideal of $\mathbf{Q}[x, y, z]$.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | (x+1)^3*y^4, x*(y-z)^2+1, z^3-z^2>;
> R := ZRadical(I);
> Groebner(I);
> Groebner(R);
> I;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0, Non-radical
Groebner basis:
[
    x - 4*y^9 + 21*y^8 - 32*y^7 + 7*y^6 + 432*y^5*z^2 - 546*y^5*z + 120*y^5 -
        137*y^4*z^2 + 288*y^4*z - 146*y^4 - 956*y^3*z^2 + 1088*y^3*z - 128*y^3 +
        393*y^2*z^2 - 576*y^2*z + 186*y^2 + 498*y*z^2 - 540*y*z + 44*y - 220*z^2
        + 288*z - 67,
    y^10 - 6*y^9 + 12*y^8 - 8*y^7 + 288*y^5*z^2 - 348*y^5*z + 60*y^5 -
        110*y^4*z^2 + 192*y^4*z - 82*y^4 - 624*y^3*z^2 + 696*y^3*z - 72*y^3 +
        273*y^2*z^2 - 384*y^2*z + 111*y^2 + 322*y*z^2 - 348*y*z + 26*y - 150*z^2
        + 192*z - 42,
    y^6*z - y^6 - 6*y^5*z^2 + 6*y^5*z - 3*y^4*z + 3*y^4 + 12*y^3*z^2 - 12*y^3*z
        + 3*y^2*z - 3*y^2 - 6*y*z^2 + 6*y*z - z + 1,
    z^3 - z^2
> R;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0, Radical
Groebner basis:
[
    x + 1,
    y^2 - 2*y*z + z - 1,
    z^2 - z
]
> I subset R;
true
> R subset I;
false
> IsInRadical(x + 1, I);
true
```

106.6.3 Relation Ideals

```
RelationIdeal(Q)
```

```
RelationIdeal(Q, T)
```

Given a sequence Q of k polynomials of a polynomial ring P over a ring S (not necessarily a field), return the relation ideal U of Q which is an ideal of the polynomial ring of rank k over S containing all algebraic relations between the elements of Q. That is, U consists of all polynomials $r \in S[y_1, \ldots, y_k]$ such that $r(Q[1], \ldots, Q[k]) = 0$. If U is desired to be an ideal of a particular polynomial ring T of rank k (to obtain predetermined names of variables, for example), then T may be passed as a second argument.

The computation is the same as that for the image of an affine polynomial map, which this basically is, thinking of the polynomials in Q as giving a map from n-dimensional affine space (n = rank of P) to k-dimensional affine space. k new variables y_i and relations $y_i - Q[i]$ are added and then the original variables x_i of P are eliminated in the usual way.

Example H106E8

We construct an ideal I of the polynomial ring $\mathbf{F}_2[x, y, z]$, and discover that the ideal is the full polynomial ring. Suppose we then wish to write $1 \in I$ as an (algebraic) expression in terms of the original generators. We use `RelationIdeal` to find that expression.

```
> P<x, y, z> := PolynomialRing(GF(2), 3, "grevlex");
> S := [(x + y + z)^2, (x^2 + y^2 + z^2)^3 + x + y + z + 1];
> I := ideal<P | S>;
> Groebner(I);
> I;
Ideal of Polynomial ring of rank 3 over GF(2)
Graded Reverse Lexicographical Order
Variables: x, y, z
Groebner basis:
[
    1
]
> Q<a, b> := PolynomialRing(GF(2), 2);
> U := RelationIdeal(S, Q);
> U;
Ideal of Polynomial ring of rank 2 over GF(2)
Order: Lexicographical
Variables: a, b
Inhomogeneous, Dimension >0
Basis:
[
    a^6 + a + b^2 + 1
]
```

Finally, we check the algebraic expression, evaluating it at the original polynomials:

```
> S[1]^6 + S[1] + S[2]^2;
1
```

106.7 Variable Extension of Ideals

Often one wishes to introduce new variables temporarily to a polynomial ring. Magma allows one to do this by use of the `VariableExtension` function, and also to restrict again to the original ring with elimination performed automatically.

`VariableExtension(I, k, b)`

`VariableExtension(I, k, b,` *order*`)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, create a polynomial ring Q as a k-variable extension of P, the ideal J of Q corresponding to I, and the embedding map $f : P \to Q$, and return J and f.

If the argument b (standing for "before") is `true`, the k variables are inserted before the current variables of P, so Q is defined to be $R[y_1, \ldots, y_k, x_1, \ldots, x_n]$ and f maps $P.i$ to $Q.(k+i)$ (so the x_i variables of P are mapped to the x_i variables of Q).

If the argument b is `false`, the k variables are inserted after the current variables of P, so Q is defined to be $R[x_1, \ldots, x_n, y_1, \ldots, y_k]$ and f maps $P.i$ to $Q.i$ (so the x_i variables of P are mapped to the x_i variables of Q).

If the argument *order* is given, then Q is constructed with the specified order; otherwise, the `grevlex` order is used for Q by default. See the section on monomial orders (Section 105.2) for the valid values for the argument *order*.

The image under f of a polynomial of P is the corresponding polynomial of Q, while the image under f of an ideal of P is the corresponding ideal of Q. The inverse image under f of a polynomial of Q is only defined if none of the extension variables of Q occur in that polynomial, in which case the inverse image is just the restriction back to P, while the inverse image under f of an ideal H of Q is always defined and is the restriction back to P of the elimination ideal $H \cap R[x_1, \ldots, x_n]$.

106.8 Homogenization of Ideals

Magma allows one to homogenize a polynomial ring or ideal by use of the `Homogenization` function, and also to restrict again to the original ring with elimination performed automatically.

`Homogenization(I, b)`

`Homogenization(I, b,` *order*`)`

`Homogenization(I)`

`Homogenization(I,` *order*`)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, create a polynomial ring H as a single variable extension of P, the homogenized ideal J of H corresponding to I, and the homogenization map $f : P \to H$, and return J and f.

If the argument b (standing for "before") is `true`, the homogenization variable is inserted before the current variables of P, so H is defined to be $R[h, x_1, \ldots, x_n]$ and f maps $P.i$ to $H.(k + i)$ (so the x_i variables of P are mapped to the x_i variables of H).

If the argument b is `false`, the homogenization variable is inserted after the current variables of P, so H is defined to be $R[x_1, \ldots, x_n, h]$ and f maps $P.i$ to $H.i$ (so the x_i variables of P are mapped to the x_i variables of H).

If the argument b is omitted, it is taken to be `false`, so the homogenization variable is introduced after the current variables of P.

If the argument *order* is given, then H is constructed with the specified order; otherwise, the `grevlex` order is used for H by default. See the section on monomial orders (Section 105.2) for the valid values for the argument *order*.

The image under f of a polynomial of P is the homogenization of f in H, while the image under f of an ideal of P is the homogenization ideal I^h in H. The inverse image under f of a polynomial of H is the restriction back to P (obtained by setting the homogenization variable to 1), while the inverse image under f of an ideal J of H is the restriction back to P of the ideal obtained by setting the homogenization variable to 1.

106.9 Extension and Contraction of Ideals

Magma allows the extension to and contraction from the ring of quotients of an ideal, defined over a field, with respect to certain variables. See [BW93, pp. 54–58 and 388–397] for the relevant definitions and theory.

`Extension(I, U)`

Given an ideal I of the polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field, together with a sequence U of integers each between 1 and n, create the (ring of quotients) extension Q of P, and return the ideal J of Q, together with the map $f : P \to Q$.

If U has length k and the values (in order) of U are $u_1, \ldots, u_k$, then first the rational function field $F = K(x_{u_1}, \ldots, x_{u_k})$ is constructed, then the list $v_1, \ldots, v_{n-k}$ is constructed as the list $1, \ldots, n$ with the u_i removed, and finally the extension Q of P is defined to be the polynomial ring $F[x_{v_1}, \ldots, x_{v_{n-k}}] = K(x_{u_1}, \ldots, x_{u_k})[x_{v_1}, \ldots, x_{v_{n-k}}]$.

The map f is constructed in the obvious way so that x_i is mapped to the appropriate variable in F if i is in U, or the appropriate variable in Q otherwise. The image under f of an ideal of P is just the appropriate ideal of Q whose basis is obtained by taking the image under f of each of the polynomials in the basis of I.

The inverse image under f of a polynomial of Q is obtained by first making the polynomial monic, then multiplying by the LCM of the denominators ("clearing the denominators"), then mapping each variable back to the appropriate one in P—this is possible since there are no proper denominators. The inverse image under f of an ideal H of Q is defined to be the ideal of P generated by the inverse images under f of the polynomials in the basis of H (note that this is not always equal to the contraction of H—see [BW93, p. 389], for a simple algorithm to compute the contraction of an ideal of Q).

106.10 Dimension of Ideals

Let I be an ideal of the polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field. Let X be the set $\{x_1, \ldots, x_n\}$ of variables of P. A subset U of X is called *independent modulo I* if $I \cap K[U] = \emptyset$. A subset U of X is called *maximally independent modulo I* if U is independent modulo I, and no proper superset of U is independent modulo I. The *dimension* of I is defined to be the maximum of the cardinalities of all the independent sets modulo I. It is not too hard to see in this case that this coincides with the more abstract commutative algebra definition of the Krull dimension of the *quotient algebra* P/I as the maximal length of a chain of prime ideals.

Note that the definition given above of zero-dimensionality (as the case when the quotient of P by I has finite dimension as a vector space over the coefficient field) coincides with the definition of zero-dimensionality as dimension 0.

```
Dimension(I)
```

Given an ideal I of a polynomial ring P defined over a field, return the dimension d of I, together with a (sorted) sequence U of integers of length d such that the variables of P corresponding to the integers of U constitute a maximally independent set modulo I. If I is the full polynomial ring P, the dimension is defined to be -1, and the second return value is not set. The algorithm implemented is that given in [BW93, p. 449].

106.11 Radical and Decomposition of Ideals

Magma has algorithms for computing the full radical and the primary decomposition of ideals. See [BW93, chapter 8], for the relevant definitions and theory. The implementation of the algorithms presented here in Magma was based on the algorithms presented in that chapter. Currently these algorithms work only for ideals of polynomial rings over fields (Euclidean rings will be supported in the future).

There are also functions for some easier decompositions than the full primary decomposition: radical decompositions, equidimensional decompositions and triangular decompositions for zero-dimensional ideals. The theory behind these is discussed in the relevant function description.

106.11.1 Radical

`Radical(I)`

Given an ideal I of a polynomial ring P over a field, return the radical of I. The radical R of I is defined to be the set of all polynomials $f \in P$ such that $f^n \in I$ for some $n \geq 1$. The radical R is also an ideal of P, containing I. The function works for an ideal defined over any field over which polynomial factorization is available.

Example H106E9

We compute the radical of an ideal of $\mathbf{Q}[x, y, z, t, u]$ (which is not zero-dimensional).

```
> P<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<P |
> x + y + z + t + u,
> x*y + y*z + z*t + t*u + u*x,
> x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
> x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
> x*y*z*t*u>;
> R := Radical(I);
> Groebner(R);
> R;
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Homogeneous, Dimension >0, Radical
Groebner basis:
[
    x + y + z + t + u,
    y^2 + y*t - z*u - u^2,
    y*z,
    y*u + z*u + u^2,
    z^2*u + z*u^2,
    z*t,
    t*u
]
```

```
> // Check that t*u is in the radical of I by another method:
> IsInRadical(t*u, I);
true
```

106.11.2 Primary Decomposition

`PrimaryDecomposition(I)`

Given an ideal I of a polynomial ring over a field, return the primary decomposition of I, and also the sequence of associated prime ideals. See `IsPrimary` for the definition of a primary ideal. The primary decomposition of I is returned as two parallel sequences Q and P of ideals, both of length k, having the following properties:

(a) The ideals of Q are primary.

(b) The intersection of the ideals of Q is I.

(c) The ideals of P are the associated primes of Q; i.e., $P[i]$ is the radical of $Q[i]$ (so $P[i]$ is prime) for $1 \leq i \leq k$.

(d) Q is minimal: no ideal of Q contains the intersection of the rest of the ideals of Q and the associated prime ideals in P are distinct.

(e) Q and P are sorted so that P is always unique and Q is unique if I is zero-dimensional. If I is not zero-dimensional, then an embedded component of Q (one whose associated prime contains another associated prime from P) will not be unique in general. Yet Magma will always return the same values for Q and P, given the same I.

The function works for an ideal defined over any field over which polynomial factorization is available (inseparable field extensions are handled by an algorithm due to Allan Steel [Ste05]).

NB: if one only wishes to compute the prime components P, then the next function `RadicalDecomposition` should be used, since this may be much more efficient.

`RadicalDecomposition(I)`

Given an ideal I of a polynomial ring over a field, return the prime decomposition of the radical of I. This is equivalent to applying the function `PrimaryDecomposition` to the radical of I, but may be a much more efficient than using that method. The (prime) radical decomposition of I is returned as a sequence P of ideals of length k having the following properties:

(a) The ideals of P are prime.

(b) The intersection of the ideals of P is the radical of I.

(c) P is minimal: no ideal of P contains the intersection of the rest of the ideals of P.

(e) P is sorted so that P is always unique. Thus Magma will always return the same values for P, given the same I.

The function works for an ideal defined over any field over which polynomial factorization is available.

`ProbableRadicalDecomposition(I)`

Given an ideal I of a polynomial ring P over a field, return a probabilistic prime decomposition of the radical of I as a sequence of ideals of P. This function is like the function `RadicalDecomposition` except that the ideals returned may not be prime, but the time taken may be much less.

`MinimalDecomposition(S)`

Given a set or sequence S of ideals of a polynomial ring over a field, with $I = \cap_{J \in S} J$ (so that S describes a decomposition of I), return a minimal decomposition M of I as a subset of S such that $I = \cap_{J \in M} J$ also (so none of the ideals in the decomposition M are redundant).

`SetVerbose("Decomposition", v)`

Change verbose printing for the (Primary/Radical) Decomposition algorithm to be v. Currently the legal values for v are `true`, `false`, 0, 1, or 2.

Example H106E10

We compute the primary decomposition of the same ideal of $\mathbf{Q}[x, y, z, t, u]$ (which is not zero-dimensional).

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

We next print out the primary components Q and associated primes P.

```
> Q;
[
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
    Groebner basis:
    [
        x + 1/2*z + 1/2*u,
        y + 1/2*z + 1/2*u,
        z^2 + 2*z*u + u^2,
        t
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
    Groebner basis:
    [
        x + 2*z + t,
        y - z,
        z^2,
        u
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
    Groebner basis:
    [
        x + z + 2*u,
        y,
        t - u,
        u^2
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
    Groebner basis:
    [
        x - u,
        y + t + 2*u,
        z,
        u^2
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
    Groebner basis:
    [
        x,
        y + 2*t + u,
        z - t,
        t^2
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
```

```
Variables: x, y, z, t, u
Homogeneous, Dimension 0, Non-radical, Primary, Non-prime
Size of variety over algebraically closed field: 1
Groebner basis:
[
    x + y + z + t + u,
    y^2 + y*t + 2*y*u - z*t + z*u + u^2,
    y*z^2 - y*z*t + y*t*u - y*u^2 + z^2*t - z^2*u + z*t*u - 2*z*u^2 + t^2*u
        + t*u^2 - u^3,
    y*z*t^2 - 2*y*z*u^2 + 3*y*t*u^2 - 2*y*u^3 + z^3*t - z^3*u - z^2*t^2 +
        4*z^2*t*u - 4*z^2*u^2 + z*t^2*u + 2*z*t*u^2 - 5*z*u^3 + 3*t^2*u^2 +
        2*t*u^3 - 2*u^4,
    y*z*t*u + y*z*u^2 - y*t^2*u - 4*y*t*u^2 + 3*y*u^3 - z^3*t + z^3*u +
        z^2*t^2 - 3*z^2*t*u + 4*z^2*u^2 - 2*z*t*u^2 + 6*z*u^3 - t^3*u -
        5*t^2*u^2 - 3*t*u^3 + 3*u^4,
    y*z*u^3 - 5/2*y*t*u^3 + 3/2*y*u^4 + 1/4*z^3*t^2 + 1/2*z^3*u^2 -
        3/4*z^2*t^3 + 5/4*z^2*t^2*u - 1/4*z^2*t*u^2 + 9/4*z^2*u^3 -
        9/4*z*t^3*u + 1/4*z*t^2*u^2 - 3/4*z*t*u^3 + 13/4*z*u^4 - t^3*u^2 -
        5/2*t^2*u^3 - 7/4*t*u^4 + 3/2*u^5,
    y*t^3*u - 17/4*y*t*u^3 + 13/4*y*u^4 + 1/8*z^3*t^2 + 5/4*z^3*u^2 -
        19/8*z^2*t^3 + 13/8*z^2*t^2*u - 5/8*z^2*t*u^2 + 33/8*z^2*u^3 -
        33/8*z*t^3*u - 7/8*z*t^2*u^2 - 31/8*z*t*u^3 + 49/8*z*u^4 + t^4*u +
        1/2*t^3*u^2 - 15/4*t^2*u^3 - 31/8*t*u^4 + 13/4*u^5,
    y*t^2*u^2 - 3/4*y*t*u^3 - 1/4*y*u^4 + 3/8*z^3*t^2 - 1/4*z^3*u^2 -
        1/8*z^2*t^3 + 7/8*z^2*t^2*u + 1/8*z^2*t*u^2 - 5/8*z^2*u^3 -
        3/8*z*t^3*u + 11/8*z*t^2*u^2 - 5/8*z*t*u^3 - 5/8*z*u^4 + 1/2*t^3*u^2
        + 3/4*t^2*u^3 - 5/8*t*u^4 - 1/4*u^5,
    y*t*u^4 - 2/3*z^2*t^4 + 13/15*z^2*t^2*u^2 - 1/5*z^2*t*u^3 -
        31/15*z*t^4*u + 3/5*z*t^3*u^2 - 2/5*z*t^2*u^3 + 23/15*z*t*u^4 -
        3/5*t^4*u^2 + 2/15*t^3*u^3 - 1/3*t^2*u^4 + t*u^5,
    y*u^5 - 1/2*z^2*t^4 - 1/2*z^2*t^2*u^2 + 1/2*z^2*t*u^3 + 1/2*z^2*u^4 -
        3/2*z*t^4*u - 3*z*t^3*u^2 + 5/2*z*t*u^4 + 3/2*z*u^5 - 1/2*t^4*u^2 -
        2*t^3*u^3 - 2*t^2*u^4 + 1/2*t*u^5,
    z^7,
    z^4*t - z^4*u - z^3*t^2 - 3*z^3*u^2 + 2*z^2*t^3 + 2*z^2*t^2*u -
        9*z^2*t*u^2 - 3*z^2*u^3 + 7*z*t^3*u + 2*z*t^2*u^2 - z*u^4 +
        2*t^3*u^2 - t^2*u^3 + t*u^4,
    z^4*u^2 + 7/3*z^2*t^4 - 40/3*z^2*t^2*u^2 + 8*z^2*t*u^3 - 3*z^2*u^4 +
        22/3*z*t^4*u - 20*z*t^3*u^2 + 2*z*t^2*u^3 + 31/3*z*t*u^4 - 2*z*u^5 +
        t^4*u^2 - 41/3*t^3*u^3 - 10/3*t^2*u^4 + 2*t*u^5,
    z^3*t^3 + 1/3*z^2*t^4 + 2/3*z^2*t^2*u^2 + z^2*t*u^3 + 1/3*z*t^4*u -
        2*z*t^3*u^2 - z*t^2*u^3 + 1/3*z*t*u^4 - 2/3*t^3*u^3 - 1/3*t^2*u^4,
    z^3*t*u - z^2*t^3 + 3*z^2*t*u^2 - 3*z*t^3*u + z*t*u^3 - t^3*u^2,
    z^3*u^3 - 1/3*z^2*t^4 + 7/3*z^2*t^2*u^2 - 2*z^2*t*u^3 + 2*z^2*u^4 -
        4/3*z*t^4*u + 7*z*t^3*u^2 - 2*z*t^2*u^3 - 13/3*z*t*u^4 + z*u^5 +
        14/3*t^3*u^3 + 4/3*t^2*u^4 - t*u^5,
    z^2*t^5 - 3*z*t*u^5 + 17/2*t^5*u^2 + 33/2*t^4*u^3 + 9*t^3*u^4 +
        15/2*t^2*u^5,
    z^2*t^3*u - z^2*t^2*u^2 + z*t^3*u^2,
    z^2*t^2*u^3 - 4/5*z*t*u^5 + 16/5*t^5*u^2 + 59/10*t^4*u^3 -
      11/10*t^3*u^4,
    z^2*t*u^4 - 4/5*z*t*u^5 + 47/10*t^5*u^2 + 42/5*t^4*u^3 - 31/10*t^3*u^4 -
      1/2*t^2*u^5,
    z^2*u^5 + 6*z*t*u^5 - 2*t^5*u^2 - 4*t^4*u^3 - 4*t^3*u^4 - 7*t^2*u^5,
    z*t^5*u + z*t*u^5 - 5/2*t^5*u^2 - 11/2*t^4*u^3 - 3*t^3*u^4 -
      5/2*t^2*u^5,
    z*t^4*u^2 + 2/5*z*t*u^5 - 11/10*t^5*u^2 - 17/10*t^4*u^3 - 1/5*t^3*u^4 -
      1/2*t^2*u^5,
    z*t^3*u^3 + 1/5*z*t*u^5 - 3/10*t^5*u^2 - 3/5*t^4*u^3 - 1/10*t^3*u^4 -
      1/2*t^2*u^5,
    z*t^2*u^4 + 2/5*z*t*u^5 - 8/5*t^5*u^2 - 16/5*t^4*u^3 - 1/5*t^3*u^4,
    t^6,
    t^5*u^3,
    t^4*u^4,
    t^3*u^5,
    u^6
  ]
]
> P;
[
  Ideal of Polynomial ring of rank 5 over Rational Field
  Order: Lexicographical
  Variables: x, y, z, t, u
  Homogeneous, Dimension 1, Radical, Prime
  Groebner basis:
  [
    x,
    y,
    z + u,
    t
  ],
  Ideal of Polynomial ring of rank 5 over Rational Field
  Order: Lexicographical
  Variables: x, y, z, t, u
  Homogeneous, Dimension 1, Radical, Prime
  Groebner basis:
  [
    x + t,
    y,
    z,
    u
  ],
  Ideal of Polynomial ring of rank 5 over Rational Field
  Order: Lexicographical
  Variables: x, y, z, t, u
  Homogeneous, Dimension >0, Radical, Prime
```

```
    Groebner basis:
    [
        x + z,
        y,
        t,
        u
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Radical, Prime
    Groebner basis:
    [
        x,
        y + t,
        z,
        u
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Radical, Prime
    Groebner basis:
    [
        x,
        y + u,
        z,
        t
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 1
    Groebner basis:
    [
        x,
        y,
        z,
        t,
        u
    ]
]
```

Notice that $P[6]$ contains other ideals of P so $Q[6]$ is an embedded primary component of I. Thus the first 5 ideals of Q would the same be in any primary decomposition of I, while $Q[6]$ could be different in another primary decomposition of I. Finally, notice that the prime decomposition of the radical of I is the same as P except for the removal of $P[6]$ to satisfy the uniqueness condition.

The structure of the variety of I can be easily understood by examining the prime decomposition of the radical.

```
> RP := RadicalDecomposition(I);
> #RP;
5
> Set(RP) eq { P[i]: i in [1 .. 5] };
true
```

106.11.3 Triangular Decomposition

Let T be a zero-dimensional ideal of the polynomial ring $K[x_1, \ldots, x_n]$, where K is a field. T is called *triangular* if its Gröbner basis G has length n and the initial term of the i-th polynomial of G is of the form $x_i^{e_i}$ for each i. Any *radical* zero-dimensional ideal has a decomposition as an intersection of triangular ideals. The algorithm in Magma for primary decomposition now (since V2.4) first computes a triangular decomposition and then decomposes each triangular component to primary ideals since the computation of a triangular decomposition is usually fast. See [Laz92] for further discussion of triangular ideals.

```
TriangularDecomposition(I)
```

Given a zero-dimensional ideal I of a polynomial ring over a field with *lexicographical* order, return a triangular decomposition of I as a sequence Q of ideals such that the intersection of the ideals of Q equals I and for each ideal J of Q which is radical, J is triangular (see above for the definition of a triangular ideal). A second return value indicates whether I is proven to be radical. If I is radical, all entries of Q are triangular. Computing a triangular decomposition will often be faster than computing the full primary decomposition and may yield sufficient information for a specific problem. The algorithm implemented is that given in [Laz92].

Example H106E11

We compute the triangular decomposition of the (radical) Cyclic-5 roots ideal and compare it with the full primary decomposition of the same ideal.

```
> R<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<R |
>     x + y + z + t + u,
>     x*y + y*z + z*t + t*u + u*x,
>     x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
>     x*y*z*t*u - 1>;
> IsRadical(I);
true
> time T := TriangularDecomposition(I);
Time: 0.000
```

```
> time Q, P := PrimaryDecomposition(I);
Time: 0.010
> #T;
9
> #Q;
20
```

So we notice that although I decomposes into 9 triangular ideals, some of these ideals must decompose further since the primary decomposition consists of 20 prime ideals. We examine the first entry of T. Notice that it is at least triangular (it has 5 polynomials and for each variable there is a polynomial whose leading monomial is a power of that variable).

```
> T[1];
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Inhomogeneous, Dimension 0
Groebner basis:
[
    x - 6/5*t^5 - 4*t^4 - 3*t^3 - 3*t^2 - 3*t - 9/5,
    y - 2/5*t^5 - 2*t^4 - 3*t^3 - 2*t^2 - 2*t - 8/5,
    z + 8/5*t^5 + 6*t^4 + 6*t^3 + 5*t^2 + 6*t + 22/5,
    t^6 + 4*t^5 + 5*t^4 + 5*t^3 + 5*t^2 + 4*t + 1,
    u - 1
]
> IsPrimary(T[1]);
false
> D := PrimaryDecomposition(T[1]);
> #D;
2
> D;
[
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Inhomogeneous, Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 2
    Groebner basis:
    [
        x - 1,
        y - 1,
        z + t + 3,
        t^2 + 3*t + 1,
        u - 1
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Inhomogeneous, Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 4
    Groebner basis:
    [
        x + t^3 + t^2 + t + 1,
        y - t^3,
        z - t^2,
        t^4 + t^3 + t^2 + t + 1,
        u - 1
    ]
]
```

106.11.4 Equidimensional Decomposition

```
EquidimensionalPart(I)
```

```
EquidimensionalDecomposition(I)
```

```
FineEquidimensionalDecomposition(I)
```

Let I be an ideal of a polynomial ring P over a field. Currently for the two decomposition functions, it is *assumed* that I has no embedded associated primes (e.g., when I is radical). In this case, it can be much faster to compute an equidimensional decomposition rather than a full primary or radical one. The equidimensional decomposition is the set of ideals which are the intersections of all primary components of I associated to primes of the same dimension. This decomposition (often trivial) is useful for certain constructions involving the Jacobian ideal.

The first function just computes the highest-dimensional decomposition component. The second performs the straight decomposition. The third gives a slightly finer decomposition for the convenience of some applications. In it, each equidimensional component is possibly further split so that, for each final equidimensional factor there is a single set of variables which constitute a maximally independent set of every primary component of the factor (*cf* `Dimension` on page 3242). A sequence of pairs consisting of each factor and the indices of its set of variables is returned.

The algorithm from [GP02] is used in the general case. When I is homogeneous, a faster, more module-theoretic method is employed for the first two functions. This involves first expressing P/I as a finite module M over a linear Noether Normalisation (described in the next section) S of I. Then if $E(I)$ is the equidimensional part of I, $E(I)/I$ as a submodule of M is equal to the kernel of the natural map of M to its double dual over S, $Hom_S(Hom_S(M,S),S)$. Working with modules over S rather than over P here allows the "reduction to dimension 0". We could directly over P, doing a similar computation but with Hom_S replaced by some Ext^i_P (see [EHV92]).

Example H106E12

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> P1 := ideal<P|x*y+y*z+z*x>; // dimension 2 prime
> P2 := ideal<P|x^2+y,y*z+2>; // dimension 1 prime
> P3 := ideal<P|x*y-1,y+z>;   // dimension 1 prime
> I := P1 meet P2 meet P3;
> time rd := RadicalDecomposition(I);
Time: 3.720
> time ed := EquidimensionalDecomposition(I);
Time: 0.070
> #ed;
2
> ed[1] eq P1;
true
> ed[2] eq (P2 meet P3);
true
```

106.12 Normalisation and Noether Normalisation

Suppose I is an ideal of $P = K[x_1, \ldots, x_n]$ with K a field, and I has dimension d.

A Noether normalisation of I is given by a set of d polynomials $f_1, \ldots, f_d$ of P, algebraically independent over K, for which $K[f_1, \ldots, f_d] \cap I = 0$ and $K[f_1, \ldots, f_d] \rightarrow P/I$ is an integral extension. These always exist and if K is an infinite field, the f_i can be chosen to be linear expressions in the x_i.

If I is radical, then the normalisation of I here will refer to the integral closure of the affine ring P/I in its total ring of fractions. If $I = \bigcap P_i$ with P_i prime, then the normalisation is equal to the finite direct product of the normalisations of the P_i as affine rings. It will be specified by a list of pairs (I_i, ϕ_i) where I_i is a prime ideal with generic ring G_i, a multivariate polynomial ring over K, and ϕ_i a homomorphism from P to G_i. The pairs represent the normalisation of each P_i and the inclusion $P/I \rightarrow \prod G_i/I_i$ induced by the ϕ_i makes the RHS the integral closure of P/I.

106.12.1 Noether Normalisation

`NoetherNormalisation(I)`

`NoetherNormalization(I)`

This function attempts to compute a Noether Normalisation for I, as described above, using *linear* combinations of the variables. The function is guaranteed to work if K has characteristic zero but may fail in unlucky cases in small characteristic.

The algorithm followed is basically that given in [GP02] but with a simpler test for homogeneous ideals I, which gives a speed-up in that case. Also, subsets of the full sets of variables are considered before more general linear combinations.

The return values are

1) the sequence $[f_1, \ldots, f_d]$.
2) h, an automorphism $P \to P$ given by a linear change of variables which maps the f_i to the last d variables of P. Thus $x_{n-d+1}, \ldots, x_n$ are a corresponding Noether normalising set of polynomials for $h(I)$.
3) the inverse of h.

Example H106E13

```
> P<x,y> := PolynomialRing(RationalField(),2);
> I := ideal<P | x*y+x+2>;
> fs,h,hinv := NoetherNormalisation(I);
> fs;
[
    x + y
]
> J := ideal<P | [h(b) : b in Basis(I)]>; J;
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: x, y
Basis:
[
    -x^2 + x*y + x + 2
]
> // clearly x is integral over the last variable y in P/J
```

106.12.2 Normalisation

`Normalisation(I)`

`Normalization(I)`

UseFF	BoolElt	*Default :* true
FFMin	BoolElt	*Default :* true
UseMax	BoolElt	*Default :* false

This function computes the normalisation of the ideal I and returns the result as a list of pairs as described above. The ideal I must be radical - this is not checked in the function. Also the base field K must be perfect.

There are several options. The general algorithm used is that of De Jong as described in [GP02]. However, if the generic polynomial ring P of I has rank 2 then Magma's powerful function field machinery can be applied to give a generally much faster algorithm. This is the default behaviour but can be bypassed by setting the parameter `UseFF` to `false`.

When the function field machinery is used, a correct result can be obtained extremely quickly, but the generic spaces of the solution ideals can be of quite high dimension. The default behaviour, controlled by the parameter FFMin, is to use a filtration by Riemann-Roch spaces to try to find a roughly minimal number of generators of the algebras G_i/I_i and return the corresponding ideal in the smaller number of variables as a more optimal presentation of the solution. This takes more time but is still usually faster than the general algorithm and tends to produce much nicer results. In some cases, the minimised solution is the same as the basic one but takes longer to generate. The minimising stage can be cut out by setting FFMin to false.

The general algorithm can avoid doing some work if it is known that certain conditions on I hold. One standard condition is that $I \subseteq M =< x_1, \dots, x_n >$ and that P/I is locally normal away from M. This holds, for example, if the affine variety defined by I in K^n is non-singular except at the origin. If this is known, then parameter UseMax can be set to true which will usually speed up the general algorithm (it has no effect if the function field method is used). However, if P/I is locally non-normal at other primes then this will produce an incorrect result.

Example H106E14

```
> P<x,y> := PolynomialRing(RationalField(),2);
> // we begin with a very simple example (prime ideal)
> I := Ideal((x - y^2)^2 - x*y^3);
> time Js := Normalisation(I); // function field method
Time: 0.010
> #Js;
1
> N := Js[1][1];
> N<[a]> := N;
> N;
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: a[1], a[2]
Basis:
[
    -a[1]*a[2] + a[2]^2 - 2*a[2] + 1
]
> // Now try the basic function field method
> time Js := Normalisation(I: FFMin:=false);
Time: 0.010
> //get the same result here either way
> N := Js[1][1];
> N<[a]> := N;
> N;
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: a[1], a[2]
Basis:
```

```
[
    -a[1]*a[2] + a[2]^2 - 2*a[2] + 1
]
> time Js := Normalisation(I:UseFF:=false); // try the general method
Time: 0.120
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: $.1, $.2, $.3, $.4
Groebner basis:
[
    $.1^2 + 2*$.1 + $.2 - 1,
    $.1*$.2 - 2*$.1 + 2*$.2 - $.3 + $.4 - 4,
    $.1*$.3 + $.3*$.4 + 2*$.3 - $.4^2,
    $.1*$.4 - $.2 + 2,
    $.2^2 - 4*$.2 - $.3*$.4 - 2*$.3 + $.4^2 + 4,
    $.2*$.3 + $.3*$.4^2 + 2*$.3*$.4 - 2*$.3 - $.4^3,
    $.2*$.4 + $.3 - 2*$.4,
    $.3^2 - $.3*$.4^3 - 2*$.3*$.4^2 + $.4^4
]
> // try the general method with UseMax (which applies here)
> time Js := Normalisation(I:UseFF:=false,UseMax:=true);
Time: 0.040
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: $.1, $.2, $.3
Groebner basis:
[
    $.1^2 - 4*$.1 - $.2*$.3 - 2*$.2 + $.3^2 + 4,
    $.1*$.2 + $.2*$.3^2 + 2*$.2*$.3 - 2*$.2 - $.3^3,
    $.1*$.3 + $.2 - 2*$.3,
    $.2^2 - $.2*$.3^3 - 2*$.2*$.3^2 + $.3^4
]
> // now try a harder case - a singular affine form of modular curve X1(11)
> I := ideal<P | (x-y)*x*(y+x^2)^3-y^3*(x^3+x*y-y^2)>;
> time Js := Normalisation(I: FFMin := false);
Time: 0.110
> #Js;
1
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 5 over Rational Field
```

```
Lexicographical Order
Variables: $.1, $.2, $.3, $.4, $.5
Groebner basis:
[
    $.1*$.3 - $.1 - 6*$.3 + $.4*$.5^2 - 4*$.4*$.5 + 6*$.4 - $.5^5 + $.5^4 +
        11*$.5^3 - 16*$.5^2 + 2*$.5 + 6,
    $.1*$.4 + 2*$.3 - $.4*$.5^2 + 2*$.4*$.5 - 2*$.4 + $.5^4 - 4*$.5^3 + 4*$.5^2
        - 2,
    $.1*$.5 - 2*$.3 + $.4 + $.5^3 - 2*$.5^2 + $.5 + 1,
    $.2 - $.3 + $.5^3 - $.5^2,
    $.3^2 + 3*$.3 - 2*$.4*$.5^2 + 4*$.4*$.5 - 4*$.4 - $.5^6 + 2*$.5^5 + $.5^4 -
        10*$.5^3 + 10*$.5^2 - 4,
    $.3*$.4 - $.3 - $.4*$.5^3 + $.4*$.5^2 - $.4*$.5 + $.4 - $.5^4 + 2*$.5^3 -
        2*$.5^2 + 1,
    $.3*$.5 + $.3 - $.4 - $.5^4 + 2*$.5^2 - $.5 - 1,
    $.4^2 - 2*$.4*$.5^2 + $.4*$.5 + $.4 - $.5^5
]
> time Js := Normalisation(I);
Time: 1.110
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: $.1, $.2
Groebner basis:
[
    $.1^2*$.2 + 2*$.1*$.2 + $.1 - $.2^2 + 2*$.2 + 1
]
> // Minimised result is a cubic equation in K[x,y] - as good as we could get!
> // This example takes MUCH longer with the general method - even setting
> // UseMax := true.
```

106.13 Hilbert Series and Hilbert Polynomial

Let I be a *homogeneous* ideal of the graded polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field. Then the quotient ring P/I is a *graded* vector space in the following way: P/I is the direct sum of the vector spaces V_d for $d = 0, 1, \ldots$ where V_d is the K-vector space consisting of all homogeneous polynomials in P/I (i.e., reduced residues of polynomials of P with respect to I) of weighted degree d. The *Hilbert Series* of the graded vector space P/I is the generating function

$$H_{P/I}(t) = \sum_{d=0}^{\infty} \dim(V_d) t^d.$$

The Hilbert series can be written as a rational function in the variable t.

If the weights on the variables of P are all 1, then there also exists the Hilbert polynomial $F_{P/I}(d)$ corresponding to the Hilbert series $H_{P/I}(t)$ which is a univariate polynomial in $\mathbf{Q}[d]$ such that $F_{P/I}(i)$ is equal to the coefficient of t^i in the Hilbert series for all $i \geq k$ for some fixed k.

```
HilbertSeries(I)
```

Given an homogeneous ideal I of a polynomial ring P over a field, return the Hilbert series $H_{P/I}(t)$ of the quotient ring P/I as an element of the univariate function field $\mathbf{Z}(t)$ over the ring of integers. The algorithm implemented is that given in [BS92].

Note that this is equivalent to `HilbertSeries(QuotientModule(I))`, while if one wishes the Hilbert series of I considered as a P-module, one should call `HilbertSeries(Submodule(I))`.

```
HilbertSeries(I, p)
```

Given an homogeneous ideal I of a polynomial ring P over a field, return the Hilbert series $H_{P/I}(t)$ of the quotient ring P/I as a power series to precision p.

```
HilbertDenominator(I)
```

Given an homogeneous ideal I of a polynomial ring P over a field, return the unreduced Hilbert denominator D of P/I (as a univariate polynomial over the ring of integers). The denominator D equals

$$\prod_{i=1}^{n}(1 - t^{w_i}),$$

where n is the rank of P and w_i is the weight of the i-th variable (1 by default).

```
HilbertNumerator(I)
```

Given an homogeneous ideal I of a polynomial ring P over a field, return the unreduced Hilbert numerator N of P/I (as a univariate polynomial over the ring of integers). The numerator N equals $D \times \times H_{P/I}(t)$, where D is the unreduced Hilbert denominator above. Computing with the unreduced numerator is often more convenient.

```
HilbertPolynomial(I)
```

Given an homogeneous ideal I of a polynomial ring P over a field with weight 1 for each variable, return the Hilbert polynomial $H(d)$ of the quotient ring P/I as an element of the univariate polynomial ring $Q[d]$, together with the index of regularity of P/I (the minimal integer $k \geq 0$ such that $H(d)$ agrees with the Hilbert function of P/I at d for all $d \geq k$).

Example H106E15

We compute the Hilbert series and Hilbert polynomial for an ideal corresponding to the square of a matrix (see [BS92]).

```
> MatSquare := function(n)
>     P := PolynomialRing(RationalField(), n * n, "grevlex");
>     AssignNames(
>         ~P,
>         ["x" cat IntegerToString(i) cat IntegerToString(j): i, j in [1..n]]
>     );
>     M := MatrixRing(P, n);
>     X := M ! [P.((i - 1) * n + j): i, j in [1 .. n]];
>     Y := X^2;
>     return ideal<P | [Y[i][j]: i, j in [1 .. n]]>;
> end function;
> I := MatSquare(4);
> I;
Ideal of Polynomial ring of rank 16 over Rational Field
Order: Graded Reverse Lexicographical
Variables: x11, x12, x13, x14, x21, x22, x23, x24, x31, x32,
    x33, x34, x41, x42, x43, x44
Homogeneous
Basis:
[
    x11^2 + x12*x21 + x13*x31 + x14*x41,
    x11*x12 + x12*x22 + x13*x32 + x14*x42,
    x11*x13 + x12*x23 + x13*x33 + x14*x43,
    x11*x14 + x12*x24 + x13*x34 + x14*x44,
    x11*x21 + x21*x22 + x23*x31 + x24*x41,
    x12*x21 + x22^2 + x23*x32 + x24*x42,
    x13*x21 + x22*x23 + x23*x33 + x24*x43,
    x14*x21 + x22*x24 + x23*x34 + x24*x44,
    x11*x31 + x21*x32 + x31*x33 + x34*x41,
    x12*x31 + x22*x32 + x32*x33 + x34*x42,
    x13*x31 + x23*x32 + x33^2 + x34*x43,
    x14*x31 + x24*x32 + x33*x34 + x34*x44,
    x11*x41 + x21*x42 + x31*x43 + x41*x44,
    x12*x41 + x22*x42 + x32*x43 + x42*x44,
    x13*x41 + x23*x42 + x33*x43 + x43*x44,
    x14*x41 + x24*x42 + x34*x43 + x44^2
]
> S<t> := HilbertSeries(I);
> S;
(t^12 - 7*t^11 + 20*t^10 - 28*t^9 + 14*t^8 + 15*t^7 - 20*t^6 +
    19*t^5 - 22*t^4 + 7*t^3 + 20*t^2 + 8*t + 1)/(t^8 - 8*t^7 +
    28*t^6 - 56*t^5 + 70*t^4 - 56*t^3 + 28*t^2 - 8*t + 1)
> H<d>, k := HilbertPolynomial(I);
> H, k;
```

```
1/180*d^7 + 7/90*d^6 + 293/360*d^5 + 61/36*d^4 + 1553/360*d^3 +
    851/180*d^2 + 101/30*d + 1
5
> // Check that evaluations of H for d >= 5 match coefficients of S:
> L<u> := LaurentSeriesRing(IntegerRing());
> L;
Laurent Series Algebra over Integer Ring
> L ! S;
1 + 16*u + 120*u^2 + 575*u^3 + 2044*u^4 + 5927*u^5 + 14832*u^6 +
    33209*u^7 + 68189*u^8 + 130642*u^9 + 236488*u^10 + 408288*u^11 +
    677143*u^12 + 1084929*u^13 + 1686896*u^14 + 2554659*u^15 +
    3779609*u^16 + 5476772*u^17 + 7789144*u^18 + 10892530*u^19 +
    O(u^20)
> Evaluate(H, 5);
5927
> Evaluate(H, 6);
14832
> Evaluate(H, 19);
10892530
```

106.14 Syzygies

The main functions to compute syzygies work with or return modules. See Chapter 109 for these. This section contains a variant that returns a basis of syzygies of a polynomial sequence as rows of a matrix.

```
SyzygyMatrix(Q)
```

Given a sequence Q of polynomials from a multivariate polynomial ring P, return the module of syzygies of Q as a matrix S. This an r by k matrix, where k is the length of Q, whose rows span the space of all vectors v such that the sum of $v[i] * Q[i]$ for $i = 1, \ldots k$ is zero. The algorithm used is the standard one, computing a module Gröbner basis with respect to a particular elimination order (see section 2.5 of [GP02], for example). The base ring may be a field or Euclidean ring.

Example H106E16

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> SyzygyMatrix([x + y, x - y, x*z + y*z]);
[           z             0            -1]
[ 1/2*x - 1/2*y -1/2*x - 1/2*y            0]
```

106.15 Maps between Rings

Magma includes functions for working with maps between multivariate polynomial rings. Let $R = K_1[x_1, \ldots, x_n]$ and $S = K_2[y_1, \ldots, y_m]$ be a polynomial rings over the fields K_1, K_2, and $f : R \to S$ a ring homomorphism.

```
PolyMapKernel(f)
```

Return the kernel of the map f as an ideal in the domain R, i.e., the set $\{a \in R | f(a) = 0\}$. This is basically the computation of the relation ideal for the polynomials defining the map and is as described in `RelationIdeal`.

```
IsInImage(f, p)
```

Given an polynomial p in S, return whether p is in the image of the map f. The algorithm is the one described on p. 82 of [AL94].

```
IsSurjective(f)
```

Return whether the map f is surjective. Uses the function above to check whether each codomain variable lies in the image.

```
Extension(phi, I)
```

The extension of the ideal I by ϕ, where ϕ is a homomorphism from the generic of I. That is, the ideal generated by the image of I under ϕ.

```
Implicitization(phi)
```

Suppose the polynomial map $\phi : K^n \to K^m$ is a parametrization of a variety V, i.e., V is the image of ϕ in K^m. This function constructs the ideal of S corresponding to V.

The map ϕ maps $(z_1, \ldots, z_n) \mapsto (f_1(z_1), \ldots, f_m(z_m))$ where the z_i are the coordinates of K^n. Let $f : S \to R$ be the map of polynomial rings defined by $(y_1, \ldots, y_m) \mapsto (f_1(y_1), \ldots, f_m(y_m))$. Then `Implicitization(f)` is the ideal of S corresponding to V.

If V is not a true variety, the function returns the smallest variety containing V (the Zariski closure of V).

The algorithm used is given on p. 97 of [CLO96]

Example H106E17

We demonstrate the use of the function `Implicitization` for the variety defined by $\phi : Q[x, y] \to Q[r, u, v, w]$, $(x, y) \mapsto (x^4, x^3y, xy^3, y^4)$. This example is taken from [AL94, Ex. 2.5.4].

```
> R<x, y> := PolynomialRing(Rationals(), 2);
> S<r, u, v, w> := PolynomialRing(Rationals(), 4);
> f := hom<S -> R |x^4, x^3*y, x*y^3, y^4>;
> Implicitization(f);
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: r, u, v, w
```

```
Basis:
[
    -r^2*v + u^3,
    r*v^2 - u^2*w,
    -u*w^2 + v^3,
    -r*w + u*v
]
```

106.16 Symmetric Polynomials

Magma includes functions for working with symmetric polynomials.

`ElementarySymmetricPolynomial(P, k)`

Given a polynomial ring P of rank n, and an integer k with $1 \leq k \leq n$, return the k-th elementary symmetric polynomial of P.

`IsSymmetric(f)`

`IsSymmetric(f, S)`

Given a polynomial f from a polynomial ring P of rank n, return whether f is a symmetric polynomial of P (i.e., is symmetric in all the n variables of P). If the answer is true, a polynomial g from a new polynomial ring of rank n is returned such that $f = g(e_1, \ldots, e_n)$, where e_i is the i-th elementary symmetric polynomial of P. If g is desired to be a member of a particular polynomial ring S of rank n (to obtain predetermined names of variables, for example), then S may also be passed.

Example H106E18

We create a symmetric polynomial from $\mathbf{Q}[a, b, c, d]$ and express it in terms of the elementary symmetric polynomials.

```
> P<a, b, c, d> := PolynomialRing(RationalField(), 4, "grevlex");
> f :=
> a^2*b^2*c*d + a^2*b*c^2*d + a*b^2*c^2*d + a^2*b*c*d^2 + a*b^2*c*d^2 +
>     a*b*c^2*d^2 - a^2*b^2*c - a^2*b*c^2 - a*b^2*c^2 - a^2*b^2*d -
>     3*a^2*b*c*d - 3*a*b^2*c*d - a^2*c^2*d - 3*a*b*c^2*d - b^2*c^2*d -
>     a^2*b*d^2 - a*b^2*d^2 - a^2*c*d^2 - 3*a*b*c*d^2 - b^2*c*d^2 -
>     a*c^2*d^2 - b*c^2*d^2 + a + b + c + d;
> // Check orbit under Sym(4) has size one:
> #(f^Sym(4));
1
> Q<e1, e2, e3, e4> := PolynomialRing(RationalField(), 4);
> l, E := IsSymmetric(f, Q);
> l;
true
> E;
```

```
e1 - e2*e3 + e2*e4
```

In the following example, we use a rational function field to define parameters a and b which occur as coefficients of the symmetric polynomial f.

```
> F<a,b> := FunctionField(RationalField(), 2);
> P<x1,x2,x3,x4,x5> := PolynomialRing(F, 5, "grevlex");
> y1 := x1^4 + x1^2*a + x1*b;
> y2 := x2^4 + x2^2*a + x2*b;
> y3 := x3^4 + x3^2*a + x3*b;
> y4 := x4^4 + x4^2*a + x4*b;
> y5 := x5^4 + x5^2*a + x5*b;
> f := y1*y2 + y1*y3 + y1*y4 + y1*y5 + y2*y3 + y2*y4 +
>      y2*y5 + y3*y4 + y3*y5 + y4*y5;
> Q<e1,e2,e3,e4,e5> := PolynomialRing(F,  5);
> l,E := IsSymmetric(f, Q);
> l, E;
true b*e1^3*e2 - 2*a*e1^3*e3 - 4*e1^3*e5 + a*e1^2*e2^2 +
    4*e1^2*e2*e4 + 2*e1^2*e3^2 - b*e1^2*e3 + 2*a*e1^2*e4 -
    4*e1*e2^2*e3 - 3*b*e1*e2^2 + 4*a*e1*e2*e3 + 8*e1*e2*e5 +
    a*b*e1*e2 - 8*e1*e3*e4 - 2*a^2*e1*e3 + b*e1*e4 - 6*a*e1*e5 +
    e2^4 - 2*a*e2^3 - 4*e2^2*e4 + a^2*e2^2 + 4*e2*e3^2 +
    5*b*e2*e3 + 2*a*e2*e4 + b^2*e2 - 3*a*e3^2 - 4*e3*e5 -
    3*a*b*e3 + 6*e4^2 + 2*a^2*e4 - 5*b*e5
```

106.17 Functions for Polynomial Algebra and Module Generators

The following functions work with collections of polynomials which are considered as generators for subalgebras or submodules of a polynomial ring. They have particular use in invariant theory.

```
MinimalAlgebraGenerators(L)
```

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose L is a set or sequence of k polynomials $f_1, \ldots, f_k$ in R. Let $A = K[f_1, \ldots, f_k]$ be the subalgebra (*not* ideal) of R generated by L. This function returns a minimal generating set of the algebra A as a (sorted) sequence of elements taken from L.

```
HomogeneousModuleTest(P, S, F)
```

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose F is an element of R. This function returns whether F is in the module M (considered as a submodule of R).

If the result is `true`, the function also returns a sequence $C = [c_1, \ldots, c_r]$ of length r with $c_i \in K[t_1, \ldots, t_k]$ such that $F = \sum_{i=1}^{r} c_i(p_1, \ldots, p_k) \cdot s_i$. (The polynomial ring $K[t_1, \ldots, t_r]$ is constructed separately but automatically with the print names `t1`, `t2`, etc.)

The grading of the polynomial ring R is used to determine the (weighted) degrees of all the polynomials in P, S and the polynomial F.

The function works as follows: it first splits F into its homogeneous components, and then, for each homogeneous component of (weighted) degree d, it constructs a basis for the K-space of all polynomials of the module M of degree d and then determines by linear algebra whether the component lies in that space.

The function is most often used with an invariant ring: P is the sequence of primary invariants, S is the sequence of secondary invariants, and F is a general invariant which one wishes to express in terms of the module generators S over the algebra generated by P. Also, if one wishes to test only for membership in the algebra $A = K[p_1, \ldots, p_k]$, then the sequence `[R!1]` should be passed for S.

```
HomogeneousModuleTest(P, S, L)
```

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose L is a sequence of length l of elements of R which are all homogeneous of (weighted) degree d. This function returns parallel sequences B and V with the following properties:

(a) B is sequence of length l of booleans such that for $1 \leq i \leq l$, $B[i]$ is true iff $L[i]$ is in the module M.

(b) V is a sequence of length l consisting of sequences of length r and consisting of polynomials in the polynomial ring $T = K[t_1, \ldots, t_r]$. (The polynomial ring $T = K[t_1, \ldots, t_r]$ is constructed separately but automatically with the print names `t1`, `t2`, etc.) If $B[i]$ is false (so $L[i]$ is not in M), $V[i]$ is a sequence of r zero polynomials. Otherwise $V[i]$ is a sequence of r polynomials $c_{i,1}, \ldots, c_{i,r}$ in T such that that $L[i] = \sum_{j=1}^{r} c_{i,j}(p_1, \ldots, p_k) \cdot s_j$.

The grading of the polynomial ring R is used to determine the (weighted) degrees of all the polynomials in P, S and L.

The function works as follows: it constructs a basis for the K-space of all polynomials of the module M of degree d and then, for each i with $1 \leq i \leq l$, determines by linear algebra whether $L[i]$ lies in the space. Only one echelonization of the space is needed to determine all the values of B and V so it is *much* more efficient to use this function if possible with many polynomials in L of the same homogeneous degree instead of calling the previous function separately for each polynomial since that will need to construct the basis for the homogeneous space and perform an echelonization each time.

Again, this function is most often used with an invariant ring: P is the sequence of primary invariants, S is the sequence of secondary invariants, and L is a sequence of general invariants which one wishes to express in terms of the module generators S over the algebra generated by P. Also, if one wishes to test only for membership in the algebra $A = K[p_1, \ldots, p_k]$, then the sequence $[R!1]$ should be passed for S.

```
HomogeneousModuleTestBasis(P, S, L)
```

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose L is a sequence of length l of elements of R which are all homogeneous of (weighted) degree d. Let U be the K-subspace of R consisting of all polynomials of the module M of (weighted) degree d and let V be the K-subspace of R generated by the elements of L. This function returns a sequence I of integer indices such that the sequence elements of L corresponding to the indices in I forms a basis for a K-subspace W of R such that $U + V = U \oplus W$. That is, I selects a subsequence of L which yields an *extension* of any basis of U to a basis of $U + V$.

Using this function, one can extend a minimal module generating set in S to include new elements of increasing degree, while ensuring that the module generators are minimalized (i.e., there is no redundancy amongst them).

The grading of the polynomial ring R is used to determine the (weighted) degrees of all the polynomials in P, S and L.

Example H106E19

We demonstrate simple uses of the function `HomogeneousModuleTest`. See also the example `HomogeneousModuleTest2` in the Invariant Rings chapter which demonstrates the use of the function `HomogeneousModuleTest` in invariant theory.

```
> R<x, y, z> := PolynomialRing(RationalField(), 3);
> P := [x^2 + y^2, z];
> S := [1, x + y + z];
> L := [x^2 + y^2, (x+y+z)^2-z^2-2*x*y, x*y];
> B, V := HomogeneousModuleTest(P, S, L);
> B;
[ true, true, false ]
```

```
> V;
[
    [
        t1,
        0
    ],
    [
        t1 - 2*t2^2,
        2*t2
    ],
    [
        0,
        0
    ]
]
> // Thus L[1] is P[1]*S[1] and
> // L[2] is (P[1] - 2*P[2]^2)*S[1] + 2*P[2]*S[2].
> L[1] eq P[1]*S[1];
true
> (P[1] - 2*P[2]^2)*S[1] + 2*P[2]*S[2] eq L[2];
true
> // Determine subsequence of [x^3, y^3, z^3] which forms
> // extension basis of module generated by P and S.
> L := [x^3, y^3, z^3];
> HomogeneousModuleTestBasis(P, S, L);
[ 1, 2 ]
> // Thus x^2 and y^2 could be appended to S to preserve
> // minimality.
```

106.18 Bibliography

[**AL94**] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases*, volume 3 of *Graduate studies in mathematics.* American Mathematical Society, Providence, R.I., 1994.

[**BS92**] David Bayer and Michael Stillman. Computation of Hilbert Functions. *J. Symbolic Comp.*, 14(1):31–50, 1992.

[**BW93**] Thomas Becker and Volker Weispfenning. *Gröbner Bases.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1993.

[**CLO96**] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms.* Undergraduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2nd edition, 1996.

[**CLO98**] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.

[**DL06**] Wolfram Decker and Christoph Lossen. *Computing in Algebraic Geometry*, volume 16 of *Algorithms and Computation in Mathematics.* Springer, New York–Berlin–Heidelberg, 2006.

[**EHV92**] D. Eisenbud, C. Huneke, and W. Vasconcelas. Direct Methods for Primary Decomposition. *Inv. math.*, 110:207–235, 1992.

[**ES94**] Eisenbud and Sturmfels. Finding sparse systems of parameters. *Journal of Pure and Applied Algebra*, 94:143–157, 1994.

[**GP02**] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra.* Springer-Verlag, Berlin–Heidelberg–New York, 2002.

[**Har77**] Robin Hartshorne. *Algebraic Geometry, GTM 52.* Springer, ASpringer, 1977.

[**Laz92**] Daniel Lazard. Solving Zero-dimensional Algebraic Systems. *J. Symbolic Comp.*, 13(2):117–131, 1992.

[**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.

107 LOCAL POLYNOMIAL RINGS

Chapter 107
LOCAL POLYNOMIAL RINGS

107.1 Introduction

This chapter describes local polynomial rings. Let R be the multivariate polynomial ring $K[x_1, \ldots, x_n]$, where K is a field. We denote by

$$K[x_1, \ldots, x_n]_{\langle x_1,\ldots,x_n \rangle}$$

the collection of all rational functions f/g of $x_1, \ldots, x_n$ with $g(p) \neq 0$, where $p = (0, \ldots, 0)$. Such a ring is **local** (has a unique maximal ideal) and we will call it a **local polynomial ring** in Magma. Such a ring is always multivariate and is related to the corresponding multivariate polynomial ring $K[x_1, \ldots, x_n]$ which will we will call **global** (when distinguishing it from the local case). We will also call $K[x_1, \ldots, x_n]_{\langle x_1,\ldots,x_n \rangle}$ the **localization** of $K[x_1, \ldots, x_n]$ (this is always understood to be at the prime ideal generated by $x_1, \ldots, x_n$, corresponding to the origin).

Much of the theory for multivariate polynomial rings and their ideals carry over to local polynomial rings, so the reader should first be familiar with multivariate polynomial rings and their ideals (see Chapters 24 and 105).

Corresponding to a Gröbner basis of an ideal of a global multivariate ring is a **standard basis** of an ideal of a local polynomial ring. See [CLO98, Chapter 4] or [GP02, Chapter 1] for the basics of the theory and algorithms.

The other facilities are currently basic but will be expanded in coming versions. But note that computations with $R-$modules, where R is a local polynomial ring, are fully supported: see Chapter 109.

107.2 Elements and Local Monomial Orders

Elements of a local polynomial ring are multivariate polynomials just like the usual (global) multivariate polynomials, except that the monomials are sorted (again with the greatest first) with respect to a **local monomial order**, which is in general the negation of a standard global monomial order. Thus the monomial 1 is less than all other monomials and the polynomials are like multivariate formal power series (written, for example, as $1 + x + x^2y + y^4$). But most arithmetic-like operations allowed for global polynomials also carry over automatically for elements of a local polynomials so we will not list them in detail in this chapter (see Chapter 24).

Note that in the strict mathematical definition of $R = K[x_1, \ldots, x_n]_{\langle x_1,\ldots,x_n \rangle}$, elements of R may have non-trivial denominators, but this is currently not supported in Magma: the elements in Magma must always be strict polynomials. The main purpose of supporting such rings is for standard bases of ideals (see below), and this restriction does not matter there, since units are automatically removed from the elements of a standard basis.

We now describe the current local monomial orders available in Magma. First the reader should see Section 105.2 for the fundamental points about (global) monomial orders for multivariate polynomial rings.

The fundamental difference in the local case is that for a local polynomial ring R of rank n, the monomial order is the negation of a global monomial order. More precisely, let M be the monomials of R. A **local monomial ordering** on M is a total order $<$ on M such that $s \leq 1$ for all $s \in M$, $s \leq t$ implies $su \leq tu$ for all $s, t, u \in M$, and M is a well-ordering (every non-empty subset of M possesses a minimal element w.r.t. $<$). See [CLO98, Sec. 4.3], [DL06, Sec. 9.1], or [GP02, Sec. 1.2] for more information.

We now list each of the monomial orders available in Magma (these will be expanded in future versions). As in the global case, we suppose that s and t are monomials from a ring R of rank n. Any order on the monomials is then fully defined by just specifying exactly when $s < t$ with respect to that order. In the following, the argument(s) are described for an order as a list of expressions; that means that the expressions (without the parentheses) should be appended to any base arguments when any particular intrinsic function is called which expects a monomial order.

107.2.1 Local Lexicographical: `llex`

Definition: $s < t$ iff there exists $1 \leq i \leq n$ such that all of the j-th exponents of s and t are equal for $i < j \leq n$, but the i-th exponent of s is greater than the i-th exponent of t. The order is specified by the argument (`"llex"`).

This order is the negation of the global lexicographical order, but with the reverse order for the variables. Thus the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable.

107.2.2 Local Graded Lexicographical: `lglex`

Definition: $s < t$ iff the total degree of s is greater than the total degree of t or the total degree of s is equal to the total degree of t and $s > t$ with respect to the (glocal) lexicographical order. The order is specified by the argument (`"lglex"`).

This order is the negation of the global `glex` order.

107.2.3 Local Graded Reverse Lexicographical: `lgrevlex`

Definition: $s < t$ iff the total degree of s is greater than the total degree of t or the total degree of s is equal to the total degree of t and $s < t$ with respect to the (global) lexicographical order applied to the exponents of s and t in reverse order. The order is specified by the argument (`"grevlex"`).

This order is the negation of the global `grevlex` order.

107.3 Local Polynomial Rings and Ideals

107.3.1 Creation of Local Polynomial Rings and Accessing their Monomial Orders

Local polynomial rings are created from a coefficient field, the number of variables, and a monomial order. If no order is specified, the monomial order is taken to be the local lexicographical order.

```
LocalPolynomialRing(K, n)
```

Create a local polynomial ring in $n > 0$ variables over the field K. The *local lexicographical* ordering on the monomials is used for this default construction.

```
LocalPolynomialRing(K, n, order)
LocalPolynomialAlgebra(K, n, order)
```

Create a local polynomial ring in $n > 0$ variables over the ring R with the given order *order* on the monomials. See the above section on local monomial orders for the valid values for the argument *order*.

```
LocalPolynomialRing(K, n, T)
```

Create a local polynomial ring in $n > 0$ variables over the field K with the order given by the tuple T on the monomials. T must be a tuple whose components match the valid arguments for the monomial orders in Section 107.2. Such a tuple is also returned by the next function.

```
MonomialOrder(R)
```

Given a local polynomial ring R (or an ideal thereof), return a description of the monomial order of R. This is returned as a tuple which matches the relevant arguments listed for each possible order in Section 107.2, so may be passed as the third argument to the function `LocalPolynomialRing` above.

```
MonomialOrderWeightVectors(R)
```

Given a polynomial ring R of rank n (or an ideal thereof), return the weight vectors of the underlying monomial order as a sequence of n sequences of n rationals. See, for example, [CLO98, p. 153] for more information.

```
Localization(R)
```

Given a (global) multivariate polynomial ring $R = K[x_1, \ldots, x_n]$ (or an ideal I of such an R), return the localization $K[x_1, \ldots, x_n]_{\langle x_1,\ldots,x_n\rangle}$ of R (or the ideal of the localization of R which corresponds to I). The print names for the variables of R are carried over.

Example H107E1

We show how one can construct local polynomial rings with different orders. Note the order on the monomials for elements of the rings.

```
> K := RationalField();
> R<x,y,z> := LocalPolynomialRing(K, 3);
> R;
Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
> MonomialOrder(R);
<"llex">
> MonomialOrderWeightVectors(R);
[
    [ 0, 0, -1 ],
    [ 0, -1, 0 ],
    [ -1, 0, 0 ]
]
> 1 + x + y + z + x^7 + x^8*y^7 + y^5 + z^10;
1 + x + x^7 + y + y^5 + x^8*y^7 + z + z^10
> R<x,y,z> := LocalPolynomialRing(K, 3, "lgrevlex");
> R;
Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Graded Reverse Lexicographical
Variables: x, y, z
> MonomialOrder(R);
<"lgrevlex">
> MonomialOrderWeightVectors(R);
[
    [ -1, -1, -1 ],
    [ -1, -1, 0 ],
    [ -1, 0, 0 ]
]
> 1 + x + y + z + x^7 + x^8*y^7 + y^5 + z^10;
1 + z + y + x + y^5 + x^7 + z^10 + x^8*y^7
```

107.3.2 Creation of Ideals and Accessing their Bases

As for global polynomial rings, within the general context of ideals of local polynomial rings, the term “basis” will refer to an *ordered* sequence of polynomials which generate an ideal. (Thus a basis can contain duplicates and zero elements so is not like a basis of a vector space.)

```
ideal< R | L >
```

Given a local polynomial ring R, return the ideal of R generated by the elements of R specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of R;

(b) A set or sequence of elements of R;

(c) An ideal of R;

(d) A set or sequence of ideals of R.

```
Ideal(B)
```

Given a set or sequence B of polynomials from a local polynomial ring R, return the ideal of R generated by the elements of B with the given basis B. This is equivalent to the above `ideal` constructor, but is more convenient when one simply has a set or sequence of polynomials.

```
Ideal(f)
```

Given a polynomial f from a local polynomial ring R, return the principal ideal of R generated by f.

```
Basis(I)
```

Given an ideal I, return the current basis of I. This will be the standard basis of I if it is computed; otherwise it will be the original basis.

```
BasisElement(I, i)
```

Given an ideal I together with an integer i, return the i-th element of the current basis of I. This the same as `Basis(I)[i]`.

107.4 Standard Bases

Computation in ideals of local polynomial rings is possible because of the construction of **standard bases** of such ideals. These are the counterpart to Gröbner bases for ideals of global polynomial rings. Currently, standard bases may only be computed for ideals defined over fields.

Magma computes a standard basis of an ideal using the Mora normal form and standard basis algorithms (with the homogenization technique): see [CLO98, Sec. 4.4] for an overview.

In contrast to the global case, for a given fixed monomial ordering a standard basis of an ideal is not unique in general because it can be difficult to get the lower order terms of polynomials in the standard basis into a unique form. But the **leading monomials** of a standard basis are always sorted in Magma and **are unique**.

107.4.1 Construction of Standard Bases

The following functions and procedures allow one to construct standard bases. Note that a standard basis for an ideal will be automatically generated when necessary; the `Groebner` procedure below simply allows control of the algorithms used to compute the standard basis. The verbose flags are shared with those for global Gröbner basis construction (since the standard basis algorithms reduce to those), so see Section 105.4.6 for details on these.

`StandardBasis(I)`

Given an ideal I, force the standard basis of I to be computed, and then return that.

`StandardBasis(S)`

Given a set or sequence S of polynomials of a local polynomial ring R, return a standard basis of the ideal generated by S as a sorted sequence.

Example H107E2

We compute the standard basis of the ideal given in [CLO98, p.167].

```
> Q := RationalField();
> R<x,y,z> := LocalPolynomialRing(Q, 3);
> I := Ideal([x^5 - x*y^6 + z^7, x*y + y^3 + z^3, x^2 + y^2 - z^2]);
> I;
Ideal of Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
Basis:
[
    x^5 - x*y^6 + z^7,
    x*y + y^3 + z^3,
    x^2 + y^2 - z^2
]
> StandardBasis(I);
[
    x^2 + y^2 - z^2,
    x*y + y^3 + z^3,
    y^3 - x*y^3 - y*z^2 - x*z^3,
    x*z^4 + 3*y^2*z^4 + 4*x*y^4*z^4 + y*z^5 + 5*x*y^3*z^5 - 2*z^6 + 4*y^2*z^6 +
        x*y^2*z^6 + z^7 - 2*x*z^7 + 7*y*z^7 + 4*x*y*z^7 - y^2*z^7 + 3*z^8 +
        4*x*z^8,
    y^2*z^4 + 3*y^5*z^5 - z^6 + 3/2*x*z^6 + 2*y^2*z^6 + y^4*z^6 - x*z^7 +
        7/2*y*z^7 - x*y^2*z^7 - y^3*z^7 + 3/2*z^8 - 3/2*x*z^8 - 2*y*z^8 +
        2*x*y*z^8 + 3/2*y^2*z^8,
    y*z^7 + 1/2*x*z^8 + 23/4*y*z^8 - 9/4*x*y^3*z^8 + 3/2*y^4*z^8,
    z^9
]
```

We note that no elements of the standard basis have factors which are units in R.

```
> [Factorization(f): f in $1];
[
    [
        <x^2 + y^2 - z^2, 1>
    ],
    [
        <x*y + y^3 + z^3, 1>
    ],
    [
        <y^2 - x*y^2 - y*z + x*y*z - x*z^2, 1>,
        <y + z, 1>
    ],
    [
        <z, 4>,
        <x + 3*y^2 + 4*x*y^4 + y*z + 5*x*y^3*z - 2*z^2 + 4*y^2*z^2 + x*y^2*z^2 +
            z^3 - 2*x*z^3 + 7*y*z^3 + 4*x*y*z^3 - y^2*z^3 + 3*z^4 + 4*x*z^4, 1>
    ],
    [
        <z, 4>,
        <y^2 + 3*y^5*z - z^2 + 3/2*x*z^2 + 2*y^2*z^2 + y^4*z^2 - x*z^3 +
            7/2*y*z^3 - x*y^2*z^3 - y^3*z^3 + 3/2*z^4 - 3/2*x*z^4 - 2*y*z^4 +
            2*x*y*z^4 + 3/2*y^2*z^4, 1>
    ],
    [
        <z, 7>,
        <y + 1/2*x*z + 23/4*y*z - 9/4*x*y^3*z + 3/2*y^4*z, 1>
    ],
    [
        <z, 9>
    ]
]
```

Example H107E3

We note that starting from an ideal I of a global polynomial ring, the standard basis of the localization of I may be much simpler than the Gröbner basis of I.

```
> Q := RationalField();
> R<x,y,z> := PolynomialRing(Q, 3);
> I := Ideal([x^2 - x*y^3 + z^3, x*y + y^2 + z, x + y^2 - z^2]);
> Groebner(I); I;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0
```

```
Groebner basis:
[
    x - y*z + 127/1052*z^9 + 585/1052*z^8 + 233/263*z^7 + 1273/1052*z^6 +
        695/526*z^5 + 223/1052*z^4 + 569/526*z^3 + 35/1052*z^2 - z,
    y^2 + y*z - 127/1052*z^9 - 585/1052*z^8 - 233/263*z^7 - 1273/1052*z^6 -
        695/526*z^5 - 223/1052*z^4 - 569/526*z^3 - 1087/1052*z^2 + z,
    y*z^2 + 51/263*z^9 + 208/263*z^8 + 308/263*z^7 + 476/263*z^6 + 465/263*z^5 +
        278/263*z^4 + 691/263*z^3 + 217/263*z^2,
    z^10 + 5*z^9 + 10*z^8 + 15*z^7 + 16*z^6 + 9*z^5 + 12*z^4 + 13*z^3 + 2*z^2
]
> QuotientDimension(I);
12
>
> IL := Localization(I);
> StandardBasis(IL);
[
    x + y^2,
    y^2 - y^3 + z,
    z^2
]
> QuotientDimension(IL);
4
```

107.5 Operations on Ideals

In the following, note that since ideals of a full polynomial ring P are regarded as subrings of P, the ring P itself is a valid ideal as well (the ideal containing 1).

107.5.1 Basic Operations

`I + J`

Given ideals I and J of the same polynomial ring P, return the sum of I and J, which is the ideal generated by the generators of I and those of J.

`I * J`

Given ideals I and J of the same polynomial ring P, return the product of I and J, which is the ideal generated by the products of the generators of I and those of J.

`I ^ k`

Given an ideal I of the polynomial ring P, and an integer k, return the k-th power of I.

`QuotientDimension(I)`

Given an ideal I of a local polynomial ring R over a field K, return the dimension of P/I as a K-vector space. Note that this is quite different from the function `Dimension` below (which returns the Krull dimension of an ideal).

`Generic(I)`

Given an ideal I of a generic local polynomial ring R, return R.

`LeadingMonomialIdeal(I)`

Given an ideal I, return the leading monomial ideal of I; that is, the ideal generated by all the leading monomials of I.

`I meet J`

Given ideals I and J of the same polynomial ring P, return the intersection of I and J.

`&meet S`

Given a set or sequence S of ideals of the same local polynomial ring R, return the intersection of all the ideals of S.

107.5.2 Ideal Predicates

`I eq J`

Given two ideals I and J of the same polynomial ring P, return whether I and J are equal.

`I ne J`

Given two ideals I and J of the same polynomial ring P, return whether I and J are not equal.

`I notsubset J`

Given two ideals I and J in the same polynomial ring P return whether I is not contained in J.

`I subset J`

Given two ideals I and J in the same polynomial ring P return whether I is contained in J.

`IsZero(I)`

Given an ideal I of the local polynomial ring R, return whether I is the zero ideal (contains zero alone).

```
IsProper(I)
```

Given an ideal I of the local polynomial ring R, return whether I is proper; that is, whether I is strictly contained in R (or whether the standard basis of I does not contain 1 alone).

```
IsZeroDimensional(I)
```

Given an ideal I of the local polynomial ring R, return whether I is zero-dimensional (so the quotient of P by I has non-zero finite dimension as a vector space over the coefficient field – see the section on dimension for further details). Note that the ring R has dimension -1, so it is not zero-dimensional.

Example H107E4

We construct some ideals in $\mathbf{Q}[x, y, z]$ and perform basic arithmetic on them.

```
> R<x,y,z> := LocalPolynomialRing(RationalField(), 3);
> I := ideal<R | x*y - z, x^3*z^2 - y^2, x*z^3 - x - y>;
> J := ideal<R | x*y - z, x^2*z - y, x*z^3 - x - y>;
> A := I * J;
> _ := StandardBasis(A);
> A;
Ideal of Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0
Standard basis:
[
    x^2 - y^2 + 2*x^3*z,
    x*y + y^2 - x^3*z,
    y^3,
    x*z + y*z,
    y*z,
    z^2
]
> M := I meet J;
> M;
Ideal of Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
Homogeneous
Basis:
[
    x + y,
    y^2,
    z
]
> A eq M;
```

```
false
> A subset M;
true
```

107.5.3 Operations on Elements of Ideals

```
f in I
```

Given a polynomial f from a local polynomial ring R, together with an ideal I of R, return whether f is in I.

```
NormalForm(f, I)
```

Given a polynomial f from a local polynomial ring R, together with an ideal I of R, return a normal form of f with respect to (the standard basis of) I. The normal form of f is zero if and only if f is in I.

```
f notin I
```

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is not in I.

Example H107E5

We demonstrate the element operations with respect to an ideal of the localization of $\mathbf{Q}[x, y, z]$.

```
> R<x,y,z> := LocalPolynomialRing(RationalField(), 3);
> I := ideal<R | (x + y)^3, (y - z)^2, y^2*z + z>;
> NormalForm(y^2*z + z, I);
0
> NormalForm(x^3, I);
-3*x^2*y
> x + y in I;
false
```

107.6 Changing Coefficient Ring

The ChangeRing function enables the changing of the coefficient ring of a local polynomial ring or ideal.

```
ChangeRing(I, L)
```

Given an ideal I of a local polynomial ring $R = K[x_1, \ldots, x_n]$ of rank n with coefficient ring K, together with a field L, construct the ideal J of the polynomial field $S = L[x_1, \ldots, x_n]$ obtained by coercing the coefficients of the elements of the basis of I into L. It is necessary that all elements of the old coefficient field K can be automatically coerced into the new coefficient field L. If K and L are fields and K is known to be a subfield of L and the current basis of I is a standard basis, then the basis of J is marked automatically to be a standard basis of J.

107.7 Changing Monomial Order

Often one wishes to change the monomial order of an ideal. Magma allows one to do this by use of the `ChangeOrder` function.

`ChangeOrder(I, Q)`

Given an ideal I of the local polynomial ring $R = K[x_1, \ldots, x_n]$, together with a local polynomial ring S of rank n (with possibly a different order to that of R), return the ideal J of S corresponding to J and the isomorphism f from R to S. The map f simply maps $R.i$ to $S.i$ for each i.

`ChangeOrder(I,` *order*`)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a monomial order *order* (see Section 107.2), construct the polynomial ring $Q = R[x_1, \ldots, x_n]$ with order *order*, and then return the ideal J of Q corresponding to I and the isomorphism f from P to Q. See the section on monomial orders for the valid values for the argument *order*. The map f simply maps $P.i$ to $Q.i$ for each i.

107.8 Dimension of Ideals

Let I be an ideal of the local polynomial ring $K[x_1, \ldots, x_n]_{\langle x_1,\ldots,x_n\rangle}$, where K is a field. As for polynomial rings, the dimension of the ideal I can be defined as the the maximum of the cardinalities of all the independent sets modulo I (see Section 107.8 for details).

`Dimension(I)`

Given an ideal I of a local polynomial ring R defined over a field, return the dimension d of I, together with a (sorted) sequence U of integers of length d such that the variables of P corresponding to the integers of U constitute a maximally independent set modulo I. If I is the full local polynomial ring R, the dimension is defined to be -1, and the second return value is not set. The algorithm implemented is that given in [BW93, p. 449].

107.9 Bibliography

[**BW93**] Thomas Becker and Volker Weispfenning. *Gröbner Bases.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1993.

[**CLO98**] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.

[**DL06**] Wolfram Decker and Christoph Lossen. *Computing in Algebraic Geometry,* volume 16 of *Algorithms and Computation in Mathematics.* Springer, New York–Berlin–Heidelberg, 2006.

[**GP02**] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra.* Springer-Verlag, Berlin–Heidelberg–New York, 2002.

108 AFFINE ALGEBRAS

Chapter 108
AFFINE ALGEBRAS

108.1 Introduction

An affine algebra in Magma is simply the quotient ring of a multivariate polynomial ring $P = R[x_1, \ldots, x_n]$ by an ideal J of P. Such rings arise commonly in commutative algebra and algebraic geometry. They can also be viewed as generalizations of number fields and algebraic function fields, when R is a field.

The elements of affine algebras are simply multivariate polynomials which are always kept reduced to normal form modulo the ideal J of "relations". Practically all operations which are applicable to multivariate polynomials are also applicable in Magma to elements of affine algebras (when meaningful).

If the ideal J of relations defining an affine algebra $A = R[x_1, \ldots, x_n]/J$ is *maximal* and R is a field, then A is a field and may be used with any algorithms in Magma which work over fields. Factorization of polynomials over such affine algebras is also supported (including fields of small characteristic, since V2.10).

If an affine algebra defined over a field has finite dimension considered as a vector space over the coefficient field, extra special operations are available on its elements.

Currently the base ring R may be a field or a Euclidean ring. Further operations for affine algebras over Euclidean rings will be supported in the future.

An affine algebra has type `RngMPolRes` and its elements type `RngMPolResElt`.

108.2 Creation of Affine Algebras

One can create an affine algebra simply by forming the quotient of a multivariate polynomial ring by an ideal (`quo` constructor or `/` function). A special constructor `AffineAlgebra` is also provided to remove the need to create the base polynomial ring.

`quo< P | J >`

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

Given a multivariate polynomial ring P and an ideal J of P, return the quotient ring P/J. The ideal J may either be specified as an ideal or by a list a_1, a_2, ..., a_r, of generators which all lie in P. The angle bracket notation can be used to assign names to the indeterminates: `Q<q, r> := quo< I | I.1 + I.2, I.2^2 - 2, I.3^2 + I.4 >;`.

`P / J`

Given a multivariate polynomial ring P and an ideal J of P, return the quotient affine algebra P/J.

```
AffineAlgebra< R, X | L >
```

Given a ring R, a list X of n identifiers, and a list L of polynomials (relations) in the n variables X, create the affine algebra of rank n with base ring R with given quotient relations; i.e., return $R[X]/\langle L\rangle$. The angle bracket notation can be used to assign names to the indeterminates.

Example H108E1

One can create a relative extension of an algebraic number field as an affine algebra. The multivariate representation will often be more efficient than an absolute representation because of the sparsity of the elements in the field.

```
> Q := RationalField();
> A<x, y> := AffineAlgebra<Q, x, y | x^2 - y^2 + 2, y^3 - 5>;
> A;
Affine Algebra of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Quotient relations:
[
    x^2 - y^2 + 2,
    y^3 - 5
]
> x^2;
y^2 - 2
> x^-1;
2/17*x*y^2 + 5/17*x*y + 4/17*x
> P<z> := PolynomialRing(Q);
> MinimalPolynomial(x);
z^6 + 6*z^4 + 12*z^2 - 17
> MinimalPolynomial(x^-1);
z^6 - 12/17*z^4 - 6/17*z^2 - 1/17
> MinimalPolynomial(y);
z^3 - 5
```

Another important construction is to create an affine algebra over a rational function field to obtain an algebraic function field:

```
> F<t> := FunctionField(IntegerRing());
> A<x, y> := AffineAlgebra<F, x, y | t*x^2 - y^2 + t + 1, y^3 - t>;
> P<z> := PolynomialRing(F);
> x^-1;
(-t^2 - t)/(t^3 + 2*t^2 + 3*t + 1)*x*y^2 - t^2/(t^3 + 2*t^2 + 3*t + 1)*x*y
    + (-t^3 - 2*t^2 - t)/(t^3 + 2*t^2 + 3*t + 1)*x
> MinimalPolynomial(x);
z^6 + (3*t + 3)/t*z^4 + (3*t^2 + 6*t + 3)/t^2*z^2 + (t^3 + 2*t^2 + 3*t +
    1)/t^3
> MinimalPolynomial(x^-1);
z^6 + (3*t^3 + 6*t^2 + 3*t)/(t^3 + 2*t^2 + 3*t + 1)*z^4 + (3*t^3 +
   3*t^2)/(t^3 + 2*t^2 + 3*t + 1)*z^2 + t^3/(t^3 + 2*t^2 + 3*t + 1)
```

In this example we can consider y as a cube root of the transcendental indeterminate t.
Note that in general the (Krull) dimension of the ideal defining the relations may be anything; it need not be 0 or 1 as it is in these examples.

108.3 Operations on Affine Algebras

This section describes operations on affine algebras. Most of the operations are very similar to those for multivariate polynomial rings; such operations are done by mapping the computation to the preimage ideal and then by mapping the result back into the affine algebra. See the corresponding functions for the multivariate polynomial rings for details.

`Q . i`

Given an affine algebra Q, return the i-th indeterminate of Q as an element of Q.

`CoefficientRing(Q)`

Return the coefficient ring of the affine algebra Q.

`Rank(Q)`

Return the rank of the affine algebra Q (the number of indeterminates of Q).

`DivisorIdeal(I)`

Given an ideal I of an affine algebra Q which is the quotient ring P/J, where P is a polynomial ring and J an ideal of P, return the ideal J.

`PreimageIdeal(I)`

Given an ideal I of an affine algebra Q which is the quotient ring P/J, where P is a polynomial ring and J an ideal of P, return the ideal I' of P such that the image of I' under the natural epimorphism $P \to Q$ is I.

`PreimageRing(Q)`

Given an affine algebra Q which is the quotient ring P/J, where P is a polynomial ring and J an ideal of P, return the polynomial ring P.

`OriginalRing(Q)`

Return the generic polynomial ring P such that Q is P/J for some ideal J of P.

`I eq J`

Given two ideals I and J of the same affine algebra Q, return `true` if and only if I and J are equal.

`I subset J`

Given two ideals I and J of the same affine algebra Q, return `true` if and only if I is contained in J.

`I + J`

Given two ideals I and J of the same affine algebra Q, return the sum $I + J$.

`I * J`

Given two ideals I and J of the same affine algebra Q, return the product $I * J$.

`I ^ n`

Given an ideal I of an affine algebra Q and an integer n, return the power I^n.

`I meet J`

Given two ideals I and J of the same affine algebra Q, return the intersection $I \cap J$.

`IsProper(I)`

Given an ideal I of the affine algebra Q, return whether I is proper; that is, whether I is strictly contained in Q.

`IsZero(I)`

Given an ideal I of the affine algebra Q, return whether I is the zero ideal. Note that this is equivalent to whether the preimage ideal of I is the divisor ideal of Q.

`IsPrime(I)`

Given an ideal I of the affine algebra Q, return whether I is a prime ideal.

`IsPrimary(I)`

Given an ideal I of the affine algebra Q, return whether I is a primary ideal.

`IsRadical(I)`

Given an ideal I of the affine algebra Q, return whether I is a radical ideal.

`PrimaryDecomposition(I)`

Given an ideal I of the affine algebra Q, return the primary decomposition of I, together with the associated primes.

`RadicalDecomposition(I)`

Given an ideal I of the affine algebra Q, return the (prime) decomposition of the radical of I.

Example H108E2

We illustrate the operations on ideals of affine algebras.

```
> Q := RationalField();
> A<x,y,z> := AffineAlgebra<Q,x,y,z | x^2 - y + 1, y^3 + z - 1>;
> A;
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
> I := ideal<A | x^3*y*z^2>;
> IsRadical(I);
false
> Radical(I);
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
Generating basis:
[
    x*y^2 + x*y - x*z + x,
    y*z,
    z^2 - z
]
> PQ, PP := PrimaryDecomposition(I);
> #PQ;
3
> PQ[1];
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
Generating basis:
[
    y + 5/81*z^3 + 1/9*z^2 + 1/3*z - 1,
    x*z^3,
    y + 5/81*z^3 + 1/9*z^2 + 1/3*z - 1,
    z^4
]
> PP[1];
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
Generating basis:
[
    x,
    y - 1,
    z
]
```

108.4 Maps between Affine Algebras

Magma includes functions for working with maps between affine algebras.

`AffineAlgebraMapKernel(phi)`

Return the kernel of the homomorphism ϕ of affine algebras.

108.5 Finite Dimensional Affine Algebras

If an affine algebra is defined over a field and has finite dimension considered as a vector space over its coefficient field, extra special operations are available on its elements.

Similar operations for affine algebras defined over general Euclidean rings will be supported in the future.

`HasFiniteDimension(Q)`

Given an affine algebra Q defined over a field, return whether Q has finite dimension.

`Dimension(Q)`

Given a finite dimensional affine algebra Q defined over a field, return the dimension of Q.

`VectorSpace(Q)`

Given a finite dimensional affine algebra Q defined over a field, construct the vector space V isomorphic to Q, and return V together with the isomorphism f from Q onto V.

`MonomialBasis(Q)`

Given a finite dimensional affine algebra Q defined over a field, return the basis B of monomials of Q. This is a sequence of monomials in Q of length d, such that that the image $f(B[i]) = V.i$ where V and f are the return values of `VectorSpace` above.

`MatrixAlgebra(Q)`

Given a finite dimensional affine algebra Q defined over a field, construct the matrix algebra A isomorphic to Q, and return A together with the isomorphism f from Q onto A.

`RepresentationMatrix(f)`

Given an element f of a finite dimensional affine algebra Q defined over a field, return the representation matrix of f, which is a d by d matrix over the coefficient field of Q (where d is the dimension of Q) which represents f.

`IsUnit(f)`

Given an element f of a finite dimensional affine algebra Q defined over a field, return whether f is a unit.

`IsNilpotent(f)`

Given an element f of a finite dimensional affine algebra Q defined over a field, return whether f is nilpotent, and if so, return also the smallest q such that $f^q = 0$.

`MinimalPolynomial(f)`

Given an element f of a finite dimensional affine algebra Q defined over a field, return the minimal polynomial of f as a univariate polynomial over the coefficient field of Q.

Example H108E3

Suppose we wish to find the minimal polynomial of $\theta = \sqrt{2} + \sqrt[3]{5}$ over $\mathbf{Q}$. To do this we can just compute the minimal polynomial of (the coset of) $x + y$ over $\mathbf{Q}$ in the affine algebra $Q[x, y]/(x^2 - 2, y^3 - 5)$.

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

108.6 Affine Algebras which are Fields

If the ideal J of relations defining an affine algebra $A = K[x_1, \ldots, x_n]/J$, where K is a field, is *maximal*, then A is a field and may be used with any algorithms in Magma which work over fields. Factorization of polynomials over such affine algebras is also supported (in any characteristic, since V2.10). The examples below will demonstrate some of the applications available.

Note that an affine algebra defined over a field which itself is a field also has finite dimension when considered as a vector space over its coefficient field, so all of the operations in the previous section are also available.

Example H108E4

We create the function field $F = \mathbf{Q}(a, b, x)$ and then the affine algebra $A = F[y]/ < y^2 - (x^3 + ax + b) >$ (which is also equivalent to an algebraic function field). This then allows us to create a generic elliptic curve E over A and compute the coordinates of multiples of a generic point easily.

```
> Q := RationalField();
> F<x, a, b> := FunctionField(Q, 3);
> A<y> := AffineAlgebra<F, y | y^2 - (x^3 + a*x + b)>;
> IsField(A);
true
> y^2;
x^3 + x*a + b
> y^-1;
1/(x^3 + x*a + b)*y
> E := EllipticCurve([A | a, b]);
> E;
Elliptic Curve defined by y^2 = x^3 + a*x + b over Affine Algebra of rank 1 over
    Rational function field of rank 3 over Rational Field
    Variables: x, a, b
> p := E ! [x, y];
> p;
(x : y : 1)
> q := 2*p;
> q;
((1/4*x^4 - 1/2*x^2*a - 2*x*b + 1/4*a^2)/(x^3 + x*a + b) : (1/8*x^6 +
5/8*x^4*a + 5/2*x^3*b - 5/8*x^2*a^2 - 1/2*x*a*b - 1/8*a^3 - b^2)/(x^6
+ 2*x^4*a + 2*x^3*b + x^2*a^2 + 2*x*a*b + b^2)*y : 1)
> c := LeadingCoefficient(q[2]);
> Denominator(c);
x^6 + 2*x^4*a + 2*x^3*b + x^2*a^2 + 2*x*a*b + b^2
> Factorization($1);
[
    <x^3 + x*a + b, 2>
]
```

Example H108E5

Starting with the same affine algebra $A = \mathbf{Q}(a, b, x)F[y]/ < y^2 - (x^3 + ax + b) >$ as in the last example, we factor some univariate polynomials over A. A is of course isomorphic to an absolute field, but the presentation given may be much more convenient to the user.

```
> Q := RationalField();
> F<x, a, b> := FunctionField(Q, 3);
> A<y> := AffineAlgebra<F, y | y^2 - (x^3 + a*x + b)>;
> P<z> := PolynomialRing(A);
> f := z^2 - (x^3 + a*x + b);
> f;
z^2 + -x^3 - x*a - b
> time Factorization(f);
[
    <z - y, 1>,
    <z + y, 1>
]
Time: 0.019
```

Example H108E6

In this final example, A is isomorphic to an algebraic number field, but its presentation may be more convenient than an absolute presentation (and may lead to sparser expressions for elements).

```
> Q := RationalField();
> A<a,b,c> := AffineAlgebra<Q, a,b,c | a^2 - b*c + 1, b^2 - c + 1, c^2 + 2>;
> P<x> := PolynomialRing(A);
> time Factorization(x^2 + 2);
[
    <x - c, 1>,
    <x + c, 1>
]
Time: 0.080
> time Factorization(x^2 - b*c + 1);
[
    <x - a, 1>,
    <x + a, 1>
]
Time: 0.090
> MinimalPolynomial(a);
x^8 + 4*x^6 + 2*x^4 - 4*x^2 + 9
> time Factorization(P ! $1);
[
    <x - a, 1>,
    <x + a, 1>,
    <x - 1/3*a*b*c - 2/3*a*b + 1/3*a*c - 1/3*a, 1>,
    <x + 1/3*a*b*c + 2/3*a*b - 1/3*a*c + 1/3*a, 1>,
    <x^4 + 2*x^2 - 2*c - 1, 1>
]
Time: 2.809
```

108.7 Rings and Fields of Fractions of Affine Algebras

Given any affine algebra $Q = K[x_1, \ldots, x_n]/J$, where K is a field, one may create the *ring of fractions* R of Q. This is the set of fractions a/b, where $a, b \in Q$ and b is invertible, and it forms a ring.

The defining ideal J does not need to be zero-dimensional. The ring of fractions R is itself represented internally by an affine algebra over an appropriate rational function field, but has the appearance to the user of the set of fractions, so one may access the numerator and denominator of elements of R, for example.

If the ideal J is prime, then R is the field of fractions of A and may be used with any algorithms in Magma which work over fields. For example, factorization of polynomials over such fields of fractions is supported (in any characteristic).

Rings of fractions have type `RngFunFrac` and their elements `RngFunFracElt`.

`RingOfFractions(Q)`

`FieldOfFractions(Q)`

Given an affine algebra Q over a field K, return the ring of fractions of Q. The only difference between the two functions is that for `FieldOfFractions`, the defining ideal of Q must be prime.

`Numerator(a)`

`Denominator(a)`

Given an element a from the ring of fractions of an affine algebra Q, return the numerator (resp. denominator) of a as an element of Q.

Example H108E7

We create the field of fractions of an affine algebra and note the basic operations.

```
> A<x,y> := AffineAlgebra<RationalField(), x,y | y^2 - x^3 - 1>;
> IsField(A);
false
> F<a,b> := FieldOfFractions(A);
> F;
Ring of Fractions of Affine Algebra of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Quotient relations:
[
    x^3 - y^2 + 1
]
```

```
> a;
a
> b;
b
> a^-1;
> a^-1;
1/(b^2 - 1)*a^2
> b^-1;
1/b
> c := b/a;
> c;
b/(b^2 - 1)*a^2
> Numerator(c);
x^2*y
> Denominator(c);
y^2 - 1
> P<X> := PolynomialRing(F);
> time Factorization(X^3 - b^2 + 1);
[
    <X - a, 1>,
    <X^2 + a*X + a^2, 1>
]
Time: 0.000
> P<X,Y> := PolynomialRing(F, 2);
> time Factorization((X + Y)^3 - b^2 + 1);
[
    <X + Y - a, 1>,
    <X^2 + 2*X*Y + a*X + Y^2 + a*Y + a^2, 1>
]
Time: 0.030
> time Factorization((b*X^2 - a)*(a*Y^3 - b + 1)*(X^3 - b^2 + 1));
[
    <Y^3 - 1/(b + 1)*a^2, 1>,
    <X - a, 1>,
    <X^2 - 1/b*a, 1>,
    <X^2 + a*X + a^2, 1>
]
Time: 0.010
```

Example H108E8

This example shows the internal operations underlying the method of constructing the field of fractions. If the ideal of relations has dimension d, then the sequence L of d maximally independent variables is passed to the extension/contraction construction, which creates a rational function field with d variables such that the ideal of relations over this field now becomes zero dimensional. Appropriate maps are set up, too.

```
> Q := RationalField();
> A<x,y> := AffineAlgebra<RationalField(), x,y | y^2 - x^3 - 1>;
> IsField(A);
false
> I := DivisorIdeal(A);
> I;
Ideal of Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Groebner basis:
[
    x^3 - y^2 + 1
]
> d, L := Dimension(I);
> d;
1
> L;
[ 2 ]
> E, f := Extension(I, L);
> E;
Ideal of Polynomial ring of rank 1 over Multivariate rational function
    field of rank 1 over Integer Ring
Graded Reverse Lexicographical Order
Variables: x
Basis:
[
    x^3 - y^2 + 1
]
> F := Generic(E)/E;
Affine Algebra of rank 1 over Multivariate rational function field of
    rank 1 over Integer Ring
Graded Reverse Lexicographical Order
Variables: x
Quotient relations:
[
    x^3 - y^2 + 1
]
> g := map<A -> F | x :-> F!f(x)>;
>
> g(x);
x
> g(y);
y
> g(x)^-1;
1/(y^2 - 1)*x^2
> g(y)^-1;
1/y
> g(x^2 + x*y);
x^2 + y*x
```

```
> g(x^2 + x*y)^-1;
y^2/(y^5 + y^4 - y^3 - 2*y^2 + 1)*x^2 + 1/(y^3 + y^2 - 1)*x - y/
    (y^3 + y^2 - 1)
> $1 * $2;
1
```

109 MODULES OVER MULTIVARIATE RINGS

Chapter 109

MODULES OVER MULTIVARIATE RINGS

109.1 Introduction

This chapter describes modules over multivariate polynomial rings and related rings. The fundamental tool for computing with such modules is the construction of Gröbner bases for modules, since these rings are not principal ideal rings in general (so standard matrix echelonization algorithms are not applicable).

In this chapter, unless otherwise indicated, a **ring** R will refer to one of the following:

(a) **Multivariate Polynomial Ring** (Chapters 24 and 105). Currently the coefficient ring of such a ring may be a field or Euclidean ring (even operations such as syzygy modules or free resolutions work over modules whose coefficient rings are Euclidean but not fields).

(b) **Local Polynomial Ring** (Localization of a Multivariate Polynomial Ring: Chapter 107; new in V2.15). Currently the coefficient ring of such a ring must be a field.

(c) **Affine Algebra** (Chapter 108). Currently the coefficient ring of such a ring must be a field.

(d) **Exterior Algebra** (Chapter 82; new in V2.15). Currently the coefficient ring of such a ring must be a field. Strictly speaking, this is a skew-commutative ring, so is not a commutative ring, and the associated modules are left R-modules, but the operations on R-modules in this chapter are practically all applicable if R is such an algebra also, so the term 'a ring R' will include such an algebra in this chapter.

In this chapter, the term "**module**" will always refer to an R-module, where R is one of the above types of ring, and such a module will have type `ModMPol` (or may have type `ModMPolGrd` if graded; see below). So we assume that the reader is generally familiar with such base rings and their ideals in Magma; see the relevant chapters for background. Many of the concepts and tools of Gröbner basis theory carry over from these types of rings.

109.2 Module Basics: Embedded and Reduced Modules

All of the modules considered in this chapter are ambient modules or embedded in such a module. We call an R-module **ambient** if it has the **explicit** presentational form $R^k/\langle \textit{relations} \rangle$, where the relations are elements of R^k (and they may be zero or not even determined, initially). Elements of an ambient R-module M are represented explicitly as vectors in R^k, and M is always generated by the k unit vectors. The **degree** of M is k.

An arbitrary module S may have a representation as a submodule of such an ambient A, which is referred to as its **ambient module**. Hence the most general definition of a module is as a sub-quotient of a free module. If A has no relations then S is just a submodule of a free module (namely, A). However, in this case, S will often also have an *internal* representation in presentational form that is essential for much of its fundamental functionality. In any case, the primary representation of elements of such an embedded module S is as vectors in the ambient.

As with vector spaces, there are two basic ways that modules can be defined in Magma: as embedded or reduced modules. A general subquotient as described above is in embedded form, but ambients may also be defined of either reduced or embedded type. The type primarily affects the way submodules and quotient modules are created. Briefly, submodules and quotient modules of embedded modules stay in embedded form (as generally proper submodules of an ambient) whereas submodules or quotients of reduced modules are always returned in presentational form as ambients, with connecting homomorphisms to link them explicitly to the original module. The two types are described in a bit more detail below. For illustration, see the examples at the end of Section 109.6.

Embedded modules are created in general via the function `EModule`, which returns a free embedded module, and in principle mimic the embedded R-spaces (as created by the function `RSpace(R, k)` in Chapter 54). Such modules are always presented with their elements and bases lying in an ambient module $R^k/\langle \mathit{relations}\rangle$. The modules are basically implemented as extensions of the multivariate polynomial ideal type (or affine algebra type if non-zero relations are present), where columns are internally added to monomials in a polynomial to represent a vector. Many operations applicable to ideals, including various Gröbner basis operations, naturally extend to such modules.

Starting with an ambient embedded module $M = R^k/\langle \mathit{relations}\rangle$, when a submodule S of M is created, the ambient module of S is still M, so the elements of S are represented as elements of R^k (modulo the relations if present); this therefore also applies to elements of any basis of S, including the Gröbner basis of S. Thus S itself may be not ambient and this is the only situation in which non-ambients can occur. Similarly, when a quotient module Q of M is constructed, the elements of Q appear as elements of R^k, while Q simply gains more relations than M, but its generators are usually not minimally reduced.

Reduced modules are created in general via the function `RModule`, which returns a free reduced module, and are more abstract and mimic the reduced modules with action over fields and Euclidean rings (as created by the function `RModule(R, k)` in Chapter 54). Such modules are **always ambient**, so always have the abstract form $R^n/\langle \mathit{relations}\rangle$, and the relationships between such modules are managed by **morphisms** lying in the background. The Gröbner basis techniques and properties are also hidden from the user in general.

Starting from a reduced module $M = R^k/\langle \mathit{relations}\rangle$, when a submodule S (having s generators $v_1, \ldots, v_s$) of M is created, S is generally created as $R^s/\langle \mathit{relations}_S\rangle$ (where the relations for S are initially unknown and are only computed when needed) and a morphism is stored from S to M, which maps the i-th unit vector of S to v_i in M. Similarly, a quotient module Q of M is constructed as another ambient module, usually with minimal generators, and a morphism from M onto Q is stored in the background. All morphisms between modules can be accessed via the function `Morphism`.

For any module M, there exists an isomorphic reduced **presentation module** P, which is always ambient, since P is reduced. If M is embedded, then P is a reduced module equivalent to M (and morphisms in the background allow automatic coercion between M and P). Otherwise, M is already reduced so P is simply identical to M. Some functions (such as `FreeResolution`) always move to the presentation of M, since it is more natural to work only with ambient modules in that context.

Embedded modules are generally preferable when one wishes to work explicitly with Gröbner bases at a very low level, while reduced modules are generally preferable for homological computations since the ambient presentation form is more convenient (particularly for the relevant maps).

Technically, there is little difference in practice between an ambient embedded module and a reduced module, if each module is considered in isolation. The concepts basically refer to how submodules and quotient modules are derived from a given module (and the fact that embedded modules allow non-ambient submodules).

Finally, there is a subclass of reduced modules with the special type `ModMPolGrd`: these are **graded**, which means that they are always generated by homogeneous elements (with respect to the relevant grading). The main distinctive of this type is simply that when one creates a submodule or quotient module of a module of type `ModMPolGrd`, then the generators must be homogeneous, thus ensuring that the new derived module is also graded so will be of type `ModMPolGrd` also. In the future, more functions will be developed which will take modules of type `ModMPolGrd` explicitly. Note also that since the type `ModMPolGrd` ISA `ModMPol` via the type 'ISA' relation, any operation applicable to a module of type `ModMPol` is also applicable to a module of type `ModMPolGrd`.

109.3 Monomial Orders

In this section we describe each of the **module monomial orders** available in Magma. If the user wishes to work with reduced modules only (particularly for homology computations), then the underlying monomial orders and Gröbner bases will probably be rarely of interest to the user, so this section may be skipped. The monomial orders are mostly of interest if one wishes to work with embedded modules with special orders so that the relevant Gröbner bases have special properties. In either case, elements of the module are represented by vectors in an ambient and we refer to the vector component positions as *columns* in analogy to matrix terminology: a presentational module is often just defined by a matrix of relations, the rows giving vectors generating the relation module and the column numbering labelling the components of the vectors. For our modules there can be a non-trivial column weighting, which we think of as applying a shift to the degree of a homogeneous polynomial that occurs as the corresponding vector component of the module element. This is used to define homogeneity and degree of the overall vector.

Given an R-module M, suppose that the underlying monomial order of R is $<_R$. A **module monomial** of M is a monomial-column pair consisting of a monomial s of R and a column number c (with $c \geq 1$), written as $s[c]$ in the following. Monomial-column pairs give an (infinite) basis for the elements in a free module R^k and a vector representing an element of M can be decomposed into a sum of scalar multiples of monomial-column pairs just as elements of the polynomial ring R can be written as a sum of scalar multiples of plain monomials.

Now suppose that $s_1[c_1]$ and $s_2[c_2]$ are module monomials from M. Any order on the pairs is then fully defined by just specifying exactly when $s_1[c_1] < s_2[c_2]$ with respect to that order. As for multivariate polynomial rings, in the following the argument(s) are described for an order as a list of expressions; that means that the expressions (without the parentheses) should be appended to any base arguments when any particular intrinsic function is called which expects a module monomial order. See [AL94, Sec. 3.5] and [CLO98, Def. 2.4] for motivation and further discussion.

109.3.1 Term Over Position: `TOP`

Definition: $s_1[c_1] < s_2[c_2]$ iff $s_1 <_R s_2$ or $s_1 = s_2$ and $c_2 > c_1$. The order is specified by the argument (`"top"`).

This order is called "TOP" (term over position) since it first compares the underlying monomials (terms with the coefficients ignored†) and then compares the columns (the positions). The column comparison is ordered so that the first column is the greatest. A Gröbner basis of a module with respect to the `TOP` order is usually the easiest to compute, and corresponds to the `grevlex` order for polynomial rings in a certain way (i.e., the order favours the 'size' of monomials and only gives priority to the columns in a secondary way).

109.3.2 Term Over Position (Weighted): `TOPW`

Definition (given a sequence W of k integer weights, where k is the degree of the ambient module): write $d_i = \mathrm{Degree}_W(s_i[c_i]) = \mathrm{Degree}(s_i) + W[c_i]$; then $s_1[c_1] < s_2[c_2]$ iff $d_1 < d_2$ or $d_1 = d_2$ and $s_1 <_R s_2$ or $d_1 = d_2$, $s_1 = s_2$ and $c_2 > c_1$. The order is specified by the arguments (`"topw", W`). The weights need not be positive (but must be small integers).

This order first compares the degrees of the monomial-coefficient pairs using both the weights of the underlying ring R and the weights on the columns given by W and then proceeds as for the `TOP` order. If there is a natural grading W on the columns of the module, then it is preferable to use this order with W, particularly if submodules of interest are homogeneous or graded w.r.t. W, since then the GB w.r.t. this order will tend to be smaller and easier to compute. Normally one would also make the base order $<_R$ to be one of the `grevlex` or `grevlexw` degree orders (see Subsections 105.2.3, 105.2.3), so that the order $<$ extends the degree order $<_R$ to a degree order on the module.

109.3.3 Position Over Term: `POT`

Definition: $s_1[c_1] < s_2[c_2]$ iff $c_2 > c_1$ or $c_1 = c_2$ and $s_1 <_R s_2$. The order is specified by the argument (`"pot"`).

This order is called "POT" (position over term) since it first compares the columns and then compares the underlying monomials. The column comparison is ordered so that the first column is the greatest. A Gröbner basis of a module with respect to the `POT` order is like an echelon form of a matrix, since the order gives priority to the columns but this is in general rather harder to compute than the GB w.r.t. the `TOP` order.

† Some authors apply the terms 'monomial' and 'term' in opposite senses to how we do here, so that is why there are the established names 'TOP' and 'POT'; we follow this instead of using 'MOP' and 'POM'!

109.3.4 Position Over Term (Permutation): `POTPERM`

Definition (given a sequence P of k integers describing a permutation of $[1..k]$, where k is the degree of the ambient module): $s_1[c_1] < s_2[c_2]$ iff $P[c_2] > P[c_1]$ or $c_1 = c_2$ and $s_1 <_R s_2$. The order is specified by the arguments (`"potperm", P`).

This order first compares the columns using the given permutation, and then compares the underlying monomials.

109.3.5 Block TOP-TOP: `TOPTOP`

Definition (given a integer k): say that a column c is in the 1st block if $c \leq k$ and in the 2nd block if $c > k$; then $s_1[c_1] < s_2[c_2]$ iff c_2 is in the 1st block and c_1 is in the 2nd block, or if the columns are in the same block and $s_1[c_1] < s_2[c_2]$ w.r.t. the `TOP` order.

This order is a block order, like an elimination order for polynomial rings: comparison is first made on the blocks in which the columns lie, and then the `TOP` order is applied within each block. A GB w.r.t. this order is easier in general to compute than the `POT` order and so is useful when one wishes to 'eliminate' the first k columns only in a GB.

109.3.6 Block TOP-POT: `TOPPOT`

Definition (given a integer k): say that a column c is in the 1st block if $c \leq k$ and in the 2nd block if $c > k$; then $s_1[c_1] < s_2[c_2]$ iff c_2 is in the 1st block and c_1 is in the 2nd block, or if the columns are in the same block and $s_1[c_1] < s_2[c_2]$ w.r.t. the `TOP`/`POT` order (respective to the 1st/2nd blocks).

This order is a block order, like an elimination order for polynomial rings: comparison is first made on the blocks in which the columns lie, and then the `TOP` order is applied within the 1st block and the `POT` order is applied within the 2nd block. This is similar to the `TOPTOP` order, but it may be preferable to order the 2nd block w.r.t. the `POT` order. Note: `POTPOT` would equal to `POT`, and `POTTOP` does not seem to be useful.

109.4 Basic Creation and Access

An ambient free module $M = R^k$ is created by giving the base ring R (see introduction above), the degree r or a sequence W of r integers for the column weights, and, optionally, an argument specifying the type of module monomial order.

109.4.1 Creation of Ambient Embedded Modules

The following functions create ambient embedded modules.

```
EModule(R, k)
```

Given a ring R, create the ambient embedded module R^k with the default `TOP` module monomial order.

```
EModule(R, k, order)
```

Given a ring R, create the ambient embedded module R^k with the module monomial order described by the given order *order*. See Section 109.3 for the valid values for *order*.

```
EModule(R, W)
```

Given a ring R and a sequence W of k integers, create the ambient embedded module R^k with column weights given by W and with the `TOPW` module monomial order with weights W.

```
EModule(R, W, order)
```

Given a ring R and a sequence W of k integers, create the ambient embedded module R^k with column weights given by W and with the module monomial order described by the given order *order*. See Section 109.3 for the valid values for *order*.

109.4.2 Creation of Reduced Modules

The following functions create reduced modules, which are always ambient.

```
RModule(R, k)
```

Given a ring R, create the reduced module R^k with zero column weights.

```
RModule(R, W)
```

Given a ring R and a sequence W of k integers, create the reduced module R^k with column weights given by W.

```
GradedModule(R, k)
```

Given a ring R, create the reduced graded module R^k with zero column weights. The resulting module has type `ModMPolGrd`, so submodules and quotient modules of it may only be generated by homogeneous elements.

Note also that in general it is preferable if possible that the base ring R has a degree ordering (such as the `grevlex` or `grevlexw` orders) so that associated Gröbner bases of derived modules will be easier to compute.

```
GradedModule(R, W)
```

Given a ring R and a sequence W of k integers, create the reduced graded module R^k with column weights given by W. The resulting module has type `ModMPolGrd`, so submodules and quotient modules of it may only be generated by homogeneous elements.

109.4.3 Localization

```
Localization(M)
```

Given an R-module M, where $R = K[x_1, \ldots, x_n]$ for a field K, return the corresponding S-module $M_{\langle x_1,\ldots,x_n\rangle}$, where $S = K[x_1, \ldots, x_n]_{\langle x_1,\ldots,x_n\rangle}$ is the localization of R. See Chapter 107 for more information.

109.4.4 Basic Invariants

The following functions access simple defining invariants of a module M.

```
Ambient(M)
Generic(M)
```

Given a module M, return the ambient (or generic) module A in which M is embedded. The only case in which A differs from M is when M is a proper submodule of an ambient embedded module. So if M is reduced, A will always equal M.

```
IsAmbient(M)
```

Given a module M, return whether M is ambient.

```
IsEmbedded(M)
```

Given a module M, return whether M is embedded.

```
IsReduced(M)
```

Given a module M, return whether M is reduced.

```
IsRoot(M)
```

Given a module M, return whether M is a root (an independent module, not derived via sub- or quotient constructions from another module).

```
CoefficientRing(M)
BaseRing(M)
```

Given an R-module M, return the base ring R over which M is defined. Note that one can then call `BaseRing(R)` to obtain the underlying ring S in which the base coefficients of elements R lie.

```
Degree(M)
```

Given an R-module M, return the degree of M, which is the k such that the ambient module of M equals $R^k/\langle \textit{relations} \rangle$. Note that if M is free and ambient, then the degree of M equals the rank of M, but otherwise in general the rank of M may be less than the degree of M (see the function `Rank` below).

```
ColumnWeights(M)
Grading(M)
```

Given a module M of degree k, return the grading of M, which is a sequence of k integers giving the grading on the columns of M.

```
RelationModule(M)
```

Given an R-module M of degree k, return the submodule of the embedded module R^k which is generated by the defining relations of M.

`Relations(M)`

Given an R-module M of degree k, return the defining relations of M as a sorted sequence of elements of the embedded module R^k.

`RelationMatrix(M)`

Given a module M, return the relation matrix of M, which is the matrix whose rows are the defining relations of M.

`Presentation(M)`

Given an R-module M, return the presentation module P of M. This is a reduced module isomorphic to M (and such that automatic coercion between M and P is allowed). If M is reduced, then P is identical to M.

`IsGraded(M)`

`IsHomogeneous(M)`

Given a module M, return whether M is graded (or equivalently, homogeneous), w.r.t. the grading of M (given by the weights on the columns of M and the variables of the base ring of M). This is true iff the Gröbner basis of M consists of homogeneous elements only (always true if M is reduced) and the Gröbner basis of the relation module of M consists of homogeneous elements alone. Note that a module of type `ModMPolGrd` is always graded.

109.4.5 Creation of Module Elements

Module elements (internally, multivariate polynomials with columns attached to the monomials) are constructed in general by giving a sequence or vector of elements from the coefficient ring R.

`M ! Q`

Suppose M is an R-module of degree r. Given a sequence $Q = [a_1, \ldots, a_r]$ of ring elements such that the a_i are coercible into R, construct the element of M corresponding to Q.

`M ! v`

Suppose M is an R-module of degree r. Given a vector v from the R-space R^r, construct the element of M corresponding to v.

`M ! 0`

`Zero(M)`

Create the zero element of the module M.

`UnitVector(M, i)`

Suppose M is an R-module of degree r. Given an integer i in the range $[1..r]$, construct the i-th unit vector of M (the vector with 1 in the i-th column and 0 elsewhere) whose parent is the ambient module of M (since it may not lie in M itself). Note that this *not* the same as the function `BasisElement` (below) which depends on the current basis of M.

109.4.6 Element Operations

The following functions allow simple access and operations on module elements. Some of them use the module structure and refer to the column structure of an element; others use the polynomial structure and ignore the column structure.

109.4.6.1 Access

`Eltseq(f)`

Given an element f of the R-module of degree r, return the sequence $[f_1, \ldots, f_r]$ of r elements from R corresponding to f.

`Vector(f)`

Given an element f of the module M over R and of degree r, return the element of the R-space of degree r over R corresponding to f.

`f[i]`

Given an element f of the R-module of degree r, together with an integer i in the range $[1..r]$, return the i-th component of f as an element of R.

109.4.6.2 Arithmetic

The following functions act on elements of R-modules. The operations are similar to those for multivariate polynomials or vectors, whenever meaningful. For the binary operations, the elements must be **compatible**; that is, their parents must have the same ambient module. Note that if quotient relations for M are present, then the result is reduced to the unique normal form modulo the quotient relations, but if the determination of the relations is delayed, then an element may have a non-unique representation, but all the predicates on elements below do not depend on the representation.

`f + g` `f - g` `- f` `r * f` `f * r`

Basic arithmetic operations. The element r lies in the base ring R.

`f div s`

Given a scalar ring element s and an element f of the module M, such that s is coercible into R s divides all components of f, return the quotient of f by s.

`SPolynomial(f, g)`

Given elements f and g of the module M such that the leading module monomials of f and g have the same column, return the S-polynomial of f and g. Note that the result is always reduced to the unique normal form modulo the quotient relations of M.

`Normalize(f)`

Given an element f of the module M, return the normalized form of f (so that the leading module monomial of f is normalized).

`NormalForm(f, S)`

Given an element f of the module M, together with a compatible module S, return the normal form of f with respect to S. This is unique if the base ring R is not local. In general, S will be a non-ambient embedded module for this to be useful (otherwise any f would already be in S so the result would always be zero).

`Coordinates(f, M)`

Given an element f of the R-module S, together with a compatible R-module M such that f is in M, return the coordinates of f with respect to the basis of M (whose components lie in R).

109.4.6.3 Accessing the Underlying Representation

The following functions access simple properties of module elements which are to do with the underlying representation.

`Coefficients(f)` `Monomials(f)` `Terms(f)`

`LeadingCoefficient(f)` `LeadingMonomial(f)`

`LeadingTerm(f)`

`CoefficientsAndMonomials(f)`

These functions are equivalent to the access functions for multivariate polynomials and access the underlying distributed polynomial representation (with columns added to the monomials); see Section 24.4.4 for details.

`Column(f)`

Given a single-term element f of a module M, return the column c of the single monomial-column pair (module monomial) $s[c]$ which f has.

`Degree(f)`

`WeightedDegree(f)`

Given an element f of a module M, return the weighted degree (abbreviated to 'degree' in this chapter) of f, which is the maximum of the weighted degrees of the monomial-column pairs of f. The weighted degree of a monomial-column $s[c]$ is the weighted degree of s (in the base ring R) plus the degree of column c in the grading of M.

`IsHomogeneous(f)`

Given an element f of a module M, return whether f is homogeneous; that is, whether the weighted degrees of all the monomial-columns of f are equal. (Note that the grading of M is thus significant.)

109.4.6.4 Predicates

`IsZero(f)`

Given an element f of the module M, return whether f is the zero element of M. Note that if the relations of M are non-zero this operation may be non-trivial (especially if the relations are not yet computed, but they will be automatically computed if needed).

`f eq g`

Given elements f and g of the module M, return whether f and g are equal. Note that this may be non-trivial (see the remarks above).

`f lt g`

Given elements f and g of the module M, return whether $f < g$ w.r.t. the underlying module monomial order. The operators `le`, `gt`, `ge` are similarly defined.

`f in M`

Given an element f of a module S together with a compatible module M, return whether f is in M.

Example H109E1

We illustrate simple modules over a multivariate polynomial ring. We construct simple ambient embedded modules over $\mathbf{Q}[x, y, z]$. The first module has default weights 0 on its columns, while the second has weights 1, 2, and 3 respectively on its columns.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3, "grevlex");
> M := EModule(R, 3);
> M;
Free Embedded Module R^3
Order: Module TOP: Graded Reverse Lexicographical
> f := M![x, y, z^2];
> g := M![z, y^3, x + 1];
> f;
[x, y, z^2]
> g;
[z, y^3, x + 1]
> f + g;
[x + z, y^3 + y, z^2 + x + 1]
> Terms(f);
[
    [0, 0, z^2],
    [x, 0, 0],
    [0, y, 0]
]
> Degree(f);
2
> [Degree(m): m in Monomials(f)];
```

```
[ 2, 1, 1 ]
> LeadingMonomial(f);
[0, 0, z^2]
> M2 := EModule(R, [10, 5, 1]);
Free Embedded Module R^3 with grading [10, 5, 1]
Order: Module TOP with column weights [10, 5, 1]: Graded Reverse Lexicographical
> f := M2![x, y, z^2];
> f;
[x, y, z^2]
> Terms(f);
[
    [x, 0, 0],
    [0, y, 0],
    [0, 0, z^2]
]
> Degree(f);
11
> [Degree(m): m in Monomials(f)];
[ 11, 6, 3 ]
```

Similar operations can be done with reduced modules. There is no difference for the elements.

```
> M := RModule(R, 3);
> M;
Free RModule R^3
> M := GradedModule(R, [10, 5, 1]);
> M;
Free Graded Module R^3 with grading [10, 5, 1]
> Grading(M);
[ 10, 5, 1 ]
> f := M![x, y^6, z^10];
> f;
[x, y^6, z^10]
> IsHomogeneous(f);
true
```

109.5 The Homomorphism Type

Magma has a special type for a homomorphism between two R-modules. The type of such a homomorphism is `ModMPolHom`. In general, functions such as `Morphism` return a homomorphism of type `ModMPolHom`, while the boundary maps of complexes are also of type `ModMPolHom` (see the function `FreeResolution`).

A homomorphism $f : M \to N$ is represented by a matrix A. There are two ways in which A can be defined:

(a) A is an **ambient** matrix: in this case, A gives the explicit map on the *ambient* modules of M and N. Thus A is $m \times n$, where $m =$`Degree(M)`, $n =$`Degree(N)`.

(b) A is a **presentation** matrix: in this case, A gives the explicit map on the *presentation* modules of M and N. Thus A is $m \times n$, where $m =$`Degree(Presentation(M))`, $n =$`Degree(Presentation(N))`.

If M and N are reduced (a common case), then they equal their respective presentation modules, so there is no difference between the above two cases (the ambient matrix and the presentation matrix are identical). So the only difference between (a) and (b) occurs when at least one of M and N is a non-ambient (proper) submodule of an embedded module.

When M and N are graded - that is, generated by elements homogeneous with respect to the ambient column weightings and with a relation module that is also generated by homogeneous elements - all homomorphisms as non-graded modules are still allowed. However there are functions to test if a given homomorphism preserves the gradings on the domain and codomain up to a constant degree shift. See `IsHomogeneous` and `Degree` below.

```
Homomorphism(M, N, A)
```

`Presentation` BoolElt *Default :* `true`

Given R-modules M and N and an $m \times n$ matrix A over R, construct the homomorphism $f : M \to N$ (with type `ModMPolHom`) defined by A.

By default, A is assumed to be a presentation matrix (see the comments above), in which case m and n must equal the degrees of the presentation modules of M and N, respectively. Alternatively, setting the parameter `Presentation` to `false` specifies that A is an ambient matrix; in this case, m and n must equal the degrees of M and N, respectively.

```
Domain(f)
```

Given a module homomorphism $f : M \to N$, return the domain M.

```
Codomain(f)
```

Given a module homomorphism $f : M \to N$, return the codomain N.

`PresentationMatrix(f)`

`Matrix(f)`

Given a module homomorphism $f : M \to N$, return the presentation matrix A_P of f as an $m \times n$ matrix corresponding to the presentation modules of M and N, respectively. This presentation matrix is always well-defined and computed, even if f is constructed via an ambient matrix.

`AmbientMatrix(f)`

`Matrix(f)`

Given a module homomorphism $f : M \to N$, return the ambient matrix A_A of f as an $m \times n$ corresponding to the ambient modules of M and N, respectively. If M and N are reduced (as commonly happens), this will be the same as the presentation matrix above. But if M and N are not reduced and f is constructed via a presentation matrix, then an error may result (since it may be impossible to give a matrix over the base ring R which gives the mapping for the ambient modules).

`f(v)`

`v * f`

Given a module homomorphism $f : M \to N$ and an element v of M, return the image of v under f, as an element of N.

`f[i]`

Given a module homomorphism $f : M \to N$ and an integer i, return the element of N corresponding to the i-th row of the ambient matrix of f.

`Image(f)`

Given a module homomorphism $f : M \to N$, return the image of f as a submodule of N (which will be reduced iff N is).

`Kernel(f)`

Given a module homomorphism $f : M \to N$, return the kernel of f as a submodule of M (which will be reduced iff M is).

`Cokernel(f)`

Given a module homomorphism $f : M \to N$, return the cokernel of f as a quotient module of N (which will be reduced iff N is).

`IsZero(f)`

Given a module homomorphism $f : M \to N$, return whether f is the zero map. Note that f may be the zero map even if the presentation or ambient matrices of f are non-zero.

`IsInjective(f)`

Given a module homomorphism $f : M \to N$, return whether f is injective (whether the kernel of f is the zero module).

`IsSurjective(f)`

Given a module homomorphism $f : M \to N$, return whether f is surjective (whether the image of f equals N).

`IsBijective(f)`

Given a module homomorphism $f : M \to N$, return whether f is bijective (injective and surjective).

`IsGraded(f)`

`IsHomogeneous(f)`

Given a module homomorphism $f : M \to N$, where M and N are graded modules, return whether f is homogeneous of some degree d; that is, whether for every pure degree element $v \in M$, $f(v) = 0$ or Degree($f(v)$) equals Degree(v) + d.

`Degree(f)`

Given a module homomorphism $f : M \to N$, return the degree of f, which is the maximum d such that an element of M of degree e is mapped via f to zero or an element of degree $e + d$. If f is homogeneous, then the 'maximum' concept is unnecessary, since the degree will be consistent for all elements of M (see the previous function).

Example H109E2

We illustrate some homomorphism functionality by looking at the explicit inclusion homomorphism between two submodules of a rank 3 free module over $\mathbf{Q}[x, y]$. We define this in non-presentational form by the identity matrix. Then we can retrieve the corresponding defining matrix for the map between the internal presentations of the two submodules. The two submodules being graded submodules, we check that the inclusion is indeed homogeneous of degree 0 (as it must be, obviously preserving degrees of elements).

```
> R<x,y> := PolynomialRing(RationalField(), 2, "grevlex");
> F := EModule(R, 3);
> // get a submodule M1 generated by a single non-zero element of F
> M1 := sub<F|[x^2,y^2,x*y]>;
> // and a second submodule M2 containing M1
> M2 := sub<F|[x,0,y],[0,y,0]>;
> incl_hm :=  Homomorphism(M1,M2,IdentityMatrix(R,3) :
>                 Presentation := false);
> incl_hm;
Module homomorphism (3 by 3)
Ambient matrix:
[1 0 0]
```

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

Now the corresponding presentation matrix of the inclusion map is the obvious one coming from the expression of the natural generator of M1 in terms of the two natural generators of M2

```
>  PresentationMatrix(incl_hm);
[y x]
> // check homogeneity of incl_hm
> IsHomogeneous(incl_hm);
true
> Degree(incl_hm);
0
```

109.6 Submodules and Quotient Modules

The following functions allow the construction of submodules and quotient modules and access to essential properties.

109.6.1 Creation

```
sub< M | L >
```

Given a module M over a ring R, return the submodule of M (with the same quotient relations as M) generated by the elements of M specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of M;

(b) A set or sequence of elements of M;

(c) A submodule of M;

(d) A set or sequence of submodules of M.

A morphism is stored from the resulting submodule S into M, such that $S.i$ is mapped to the i-th generator given in the above list.

```
quo< M | L >
```

Given a module M over a ring R, return the quotient module of M by the elements of M specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of M;

(b) A set or sequence of elements of M;

(c) A submodule of M;

(d) A set or sequence of submodules of M.

A morphism is stored from M onto the resulting quotient module Q.

`Morphism(M, N)`

Given modules M and N, related by a chain of stored sub and quo morphisms as mentioned above, returns the resulting morphism matrix map from M to N. If no known sub/quo relationship chain exists between M and N then an error is returned.

`Submodule(I)`

Given an ideal I of a polynomial ring R, return the submodule of R^1 generated by I.

`QuotientModule(I)`

Given an ideal I of a polynomial ring R, return the quotient module R^1/I.

`GradedModule(I)`

Given a homogeneous ideal I of a ring R, return the graded quotient module R^1/I.

109.6.2 Module Bases

The following functions allow one to manipulate the bases of modules. Note that a Gröbner basis for a module will be automatically generated when necessary; the `Groebner` procedure just allows explicit immediate construction of the Gröbner basis.

`Basis(M)`

Given a module M, return the current basis (whether it has been converted to a Gröbner basis or not) of M.

`BasisElement(M, i)`

Given a module M together with an integer i, return the i-th element of the current basis of M. Note that this is *not* the same as $M.i$.

`BasisMatrix(M)`

Given a module M, return the basis matrix of M, which is a k by r matrix over R, where k is the length of the basis of M and r is the degree of M.

`Groebner(M)`

(Procedure.) Explicitly force a Gröbner basis for the module M to be constructed.

Example H109E3

We construct simple submodules and quotient modules of an embedded module and consider some of their basic properties.

```
> R<x, y, z> := PolynomialRing(RationalField(), 3);
> M := EModule(R, 3);
> S := sub<M | [1, x, x^2+y], [z, y, x*y^2+1],
>              [y, z, x+z]>;
> Groebner(S);
> S;
Embedded Submodule of R^3
Order: Module TOP: Lexicographical
Groebner basis:
[ -x*z + y^2 + y, x*y^2 - x*y + z,         y^3 + z],
[  x*y - y*z - 1,   x*z - x - z^2,        -y - z^2],
[              y,               z,           x + z],
[        y^3 - z,       y^2*z - y,       y^2*z - 1]
> a := M ! [y, z, x+z];
> a;
[y, z, x + z]
> a in S;
true
> BasisElement(S, 1);
[-x*z + y^2 + y, x*y^2 - x*y + z, y^3 + z]
> Q := quo<M | [x, y, z]>;
> Q;
Embedded Module R^3/<relations>
Order: Module TOP: Lexicographical
Relations (Groebner basis):
[x, y, z]
> a := Q![x, y, 0];
> b := Q![0, 0, z];
> a;
[0, 0, -z]
> b;
[0, 0, z]
> a+b;
[0, 0, 0]
> Q ! [x,y,z];
[0, 0, 0]
> QQ := quo<Q | [x^2, 0, y+z]>;
> QQ;
Embedded Module R^3/<relations>
Order: Module TOP: Lexicographical
Relations (Groebner basis):
[          0,         x*y, x*z - y - z],
[          x,           y,           z]
> SL := Localization(S);
> SL;
Embedded Submodule of R^3 (local)
Order: Module TOP: Local Lexicographical
Basis:
[        1,         x,   x^2 + y],
[        z,         y, 1 + x*y^2],
[        y,         z,     x + z]
```

Example H109E4

We construct simple submodules and quotient modules of a reduced module and consider some of their basic properties.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> M := RModule(R, 3);
> S := sub<M | [1, x, x^2+y], [z, y, x*y^2+1]>;
> M;
Free Reduced Module R^3
> S;
Reduced Module R^2/<relations>
> Morphism(S, M);
Module homomorphism (2 by 3)
Ambient matrix:
[        1         x   x^2 + y]
[        z         y x*y^2 + 1]
> RelationMatrix(S);
Matrix with 0 rows and 2 columns
> S;
Free Reduced Module R^2
> M.1;
[1, 0, 0]
> M!S.1;
[1, x, x^2 + y]
> M!S.2;
[z, y, x*y^2 + 1]
> M.1 in S;
false
> Q := quo<M | [1, x^2, y]>;
> Q;
Free Reduced Module R^2
> RelationMatrix(Q);
Matrix with 0 rows and 2 columns
> Morphism(M, Q);
Module homomorphism (3 by 2)
Ambient matrix:
[-x^2   -y]
[   1    0]
[   0    1]
```

```
> Morphism(S, Q);
Module homomorphism (2 by 2)
Ambient matrix:
[        -x^2 + x              x^2]
[      -x^2*z + y x*y^2 - y*z + 1]
> Q!M.1;
[-x^2, -y]
> M!Q.1;
[0, 1, 0]
> M!Q.2;
[0, 0, 1]
> Q!M!Q.2;
[0, 1]
```

109.7 Basic Module Constructions

The following functions give some fundamental basic constructions with modules.

`M + N`

Given compatible modules M and N (ie, embedded in the same ambient module), return the sum of M and N; that is, the submodule of the ambient generated by M and N.

`M meet N`

Given compatible modules M and N (ie, embedded in the same ambient module), return the intersection of M and N in the ambient. This uses the standard algorithm for intersecting two modules of a free module (see Section 2.8.3 of [GP02]). If the ambient is the quotient of a free module F by non-trivial relations, the intersection performed is effectively that of the inverse images of M and N in F.

`f * M`

`M * f`

Given an R-module M and an element $f \in R$, return the submodule of M generated by $\{f \cdot v : v \in M\}$ or $\{v \cdot f : v \in M\}$, respectively.

`I * M`

`M * I`

Given an R-module M and an ideal I of R, return the submodule of M generated by $\{f \cdot v : f \in I, v \in M\}$ or $\{v \cdot f : f \in I, v \in M\}$, respectively.

`M / N`

Given compatible modules M and N (ie, embedded in the same ambient module), return the quotient module $M/(M \cap N)$. This has the same effect as using the `quo` constructor.

```
DirectSum(M, N)
```

Given R-modules M and N, return the direct sum $D = M \oplus N$ and two sequences of corresponding homomorphisms giving the injections into and projections from D, respectively.

```
DirectSum(S)
```

A sequence or list L of R-modules, return their direct sum D and two sequences of corresponding homomorphisms giving the injections into and projections from D, respectively.

```
Twist(M, d)
```

Given a graded module M, and an integer d, return the Serre twist $M(d)$ and an isomorphism $f : M \to M(d)$. The twisted module is simply an isomorphic copy of M, but with the grading twisted by d (so d is subtracted from each weight of M). f has degree $-d$.

109.8 Predicates

```
IsZero(M)
```

Given a module M, return whether M is the zero module.

```
M subset N
```

Given compatible modules M and N (ie, embedded in the same ambient module), return whether M is a submodule of N. This will generally involve module Gröbner basis and normal form computations to check that the generators of M lie in N.

```
M eq N
```

Given compatible modules M and N (ie, embedded in the same ambient module), return whether M equals N. The function checks that appropriate module Gröbner bases of M and N are equal.

```
IsFree(M)
```

Given an R-module M, return whether M is free. M is free iff M is isomorphic to the module R^k for some k. Such a k need not equal the degree of M but will equal the rank of M (as defined in the next section) if M is free. The function checks whether a minimised presentation of M has trivial relations or not.

109.9 Module Operations

The following functions perform some fundamental module operations.

`MinimalBasis(M)`

Given an R-module M, return a minimal basis B of M. If M is graded, or if R is a local ring, then the cardinality of B (the rank) is guaranteed to be unique (so is the absolutely minimal number of elements needed to generate M).

Otherwise the cardinality of B is not unique: B will only satisfy the rule that the i-th element of B is not in the submodule generated by elements 1 to $i-1$ of B.

In the graded case or local cases, a minimal basis is computed in the usual way starting from any basis B consisting of homogeneous elements. B gives a particular presentation whose relation matrix R consists of homogeneous polynomials. If R contains a non-zero constant term (or more generally a unit in the local case), an element of B can be eliminated and R recalculated. This can be continued until all non-zero terms of R have positive degree.

`MinimalBasis(S)`

Given a set or sequence S of homogeneous module elements from a module M, return a minimal basis of the submodule of M generated by S.

`Rank(M)`

Given an R-module M, return the rank of M. This is simply defined to be the cardinality of the minimal basis of M, returned by the function `MinimalBasis`. Thus if M is graded, or if R is a local ring, then the rank is guaranteed to be unique (and is the absolutely minimal number of elements needed to generate M). Otherwise the result is not an invariant of M, but simply reflects the minimum as found by the `MinimalBasis` algorithm.

`ColonModule(M, J)`

Given an R-module M and an ideal J of R, return the colon module $M : J$ which is the submodule of the ambient module A of M consisting of all $f \in A$ such that $f \cdot g \in M$ for all $g \in J$. When J is generated by a single element, this easily reduces to a syzygy computation in A and in the general case, we intersect the colon modules for a set of generators of J.

`ColonIdeal(M, N)`

Given an R-modules M and N which are both submodules of a common supermodule, return the colon ideal $M : N$, which is the ideal of R consisting of all $f \in R$ such that $f \cdot N \subset M$. The algorithm used is as described in section 2.8.4 of [GP02].

`Annihilator(M)`

Given an R-module M, return the annihilator ideal of M. This is the ideal I of R consisting of all $f \in R$ such that $f \cdot M = 0$ (which can be seen to equal the ideal $0_M : M$, where 0_M is the zero submodule of M, so is a special case of `ColonIdeal`).

`FittingIdeal(M, i)`

Given an R-module M of degree r and an integer $i \geq 0$, return the i-th Fitting ideal of M, which is the ideal of R generated by the $(r-i)$-th minors of the presentation matrix of M, where r is the degree of M. See [CLO98, p.229] or [Eis95, Sec. 20.2].

`FittingIdeals(M)`

Given an R-module M of degree r, return the Fitting ideals (for from 0 to r) as a sequence of ideals of R.

`SyzygyModule(M)`

Given a module M, return the syzygy module S of M. If the basis B of M has length k, the syzygy module S has degree k and elements of S express a syzygy amongst the k elements of the basis B. Note that the degree of the resulting module thus depends on the current basis of M.

`MinimalSyzygyModule(M)`

Given a homogeneous module M, return the syzygy module S of the minimal basis of M. If the minimal basis B of M has length k, the syzygy module S has degree k and elements of S express a syzygy amongst the k elements of the minimal basis B.

`SyzygyModule(Q)`

Given a sequence Q of polynomials from a multivariate polynomial ring P, return the module of syzygies of Q. This is a module over P of degree k, where k is the length of Q, consisting of all vectors v such that the sum of $v[i] * Q[i]$ for $i = 1, \dots k$ is zero.

Example H109E5

In this example we note that a certain module M has rank 3 (equal to its degree 3), since no generator is redundant. If we move to the localization of M, then $(1 + x - z)$ becomes a unit, so the first generator becomes redundant.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3, "grevlex");
> F := RModule(R, 3);
> M := quo<F | [x + 1, y, z], [z, y, 0]>;
> M;
Reduced Module R^3/<relations>
Relations:
[x + 1,     y,     z],
[    z,     y,     0]
> Degree(M);
3
> Rank(M);
3
> ML := Localization(M);
> ML;
```

```
Reduced Module R^3/<relations> (local)
Relations:
[1 + x - z,         0,         z],
[        z,         y,         0]
> Rank(ML);
2
> MinimalBasis(ML);
[
    [0, 1, 0],
    [0, 0, 1]
]
```

109.10 Changing Ring

The `ChangeRing` function enables the changing of the polynomial ring over which a module is defined.

`ChangeRing(M, S)`

Given an R-module M, where R is a polynomial ring, and another polynomial ring S, construct the S-module N obtained by coercing the coefficients of the elements of the basis and relations of M into S. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S. Note that S itself must be polynomial ring having the same rank as R, so S does not specify the new ring for the underlying coefficients (one can use `ChangeRing` for polynomial rings to do that first).

109.11 Hilbert Series

The following functions compute the Hilbert series information of graded or (homogeneous) modules. This depends on the column weights, just as in graded polynomial rings.

`HilbertSeries(M)`

Given a graded R-module M, return the Hilbert series $H_M(t)$ of M (as a univariate function field over the ring of integers. The i-th coefficient of the series gives the vector-space dimension of the degree-i graded piece of M. The algorithm implemented is that given in [BS92].

Note that if I is an ideal of the ring R, then the corresponding function for ideals `HilbertSeries` applied to I gives the Hilbert series of the affine algebra (quotient) R/I, so this is equivalent to `HilbertSeries(QuotientModule(I))`.

`HilbertSeries(M, p)`

Given a graded R-module M, return the Hilbert series $H_M(t)$ of M as a Laurent series to precision p. (A Laurent series is required in general, since negative powers may occur when there are negative values in the grading of M.)

`HilbertDenominator(M)`

Given a graded R-module M, return the unreduced Hilbert denominator D of the Hilbert series $H_M(t)$ of M (as a univariate polynomial over the ring of integers). The denominator D equals `HilbertDenominator(R)` which is simply

$$\prod_{i=1}^{n}(1 - t^{w_i}),$$

where n is the rank of R and w_i is the weight of the i-th variable (1 by default).

`HilbertNumerator(M)`

Given a graded R-module M, return the unreduced Hilbert numerator N of the Hilbert series $H_M(t)$ of M (as a univariate polynomial over the ring of integers) and a valuation shift s. The numerator N equals $D \times t^s \times H_M(t)$, where D is the unreduced Hilbert denominator above. Computing with the unreduced numerator is often more convenient. Note that s will only be non-zero when M has negative weights in its grading.

`HilbertPolynomial(I)`

Given a graded R-module M, return the Hilbert polynomial $H(d)$ of M as an element of the univariate polynomial ring $\mathbf{Q}[d]$, together with the index of regularity of M (the minimal integer $k \geq 0$ such that $H(d)$ agrees with the Hilbert function of M at d for all $d \geq k$).

Example H109E6

We apply the Hilbert series functions to a simple quotient module.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> F := GradedModule(R, 3);
> M := quo<F | [x,0,0], [0,y^2,0]>;
> M;
Graded Module R^3/<relations>
Relations:
[  x,   0,   0],
[  0, y^2,   0]
> HilbertSeries(M);
(t^2 + t - 3)/(t^3 - 3*t^2 + 3*t - 1)
> HilbertSeries(M, 10);
3 + 8*s + 14*s^2 + 21*s^3 + 29*s^4 + 38*s^5 + 48*s^6 + 59*s^7 + 71*s^8 + 84*s^9
    + O(s^10)
> HilbertNumerator(M);
-x^2 - x + 3
0
> HilbertDenominator(M);
-x^3 + 3*x^2 - 3*x + 1
```

```
> HilbertPolynomial(M);
1/2*x^2 + 9/2*x + 3
0
> [Evaluate(HilbertPolynomial(F), i): i in [0..10]];
[ 3, 9, 18, 30, 45, 63, 84, 108, 135, 165, 198 ]
```

If the module has negative weights, then denominator may include extra powers of t, so the shift for the numerator will be non-zero.

```
> F := GradedModule(R, [-1]);
> F;
Free Graded Module R^1 with grading [-1]
> HilbertSeries(F);
-1/(t^4 - 3*t^3 + 3*t^2 - t)
> HilbertSeries(F, 10);
s^-1 + 3 + 6*s + 10*s^2 + 15*s^3 + 21*s^4 + 28*s^5 + 36*s^6 + 45*s^7 + O(s^8)
> HilbertNumerator(F);
1
1
> HilbertDenominator(F);
-x^3 + 3*x^2 - 3*x + 1
> HilbertPolynomial(F);
1/2*x^2 + 5/2*x + 3
-1
> [Evaluate(HilbertPolynomial(F), i): i in [-1..10]];
[ 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78 ]
```

109.12 Free Resolutions

The functions in this section deal with free resolutions and associated properties. Free resolutions are returned as chain complexes (see Chapter 56).

109.12.1 Constructing Free Resolutions

`FreeResolution(M)`

`Minimal`	BoolElt	*Default :* `true`
`Limit`	RngIntElt	*Default :* 0
`Homogenize`	BoolElt	*Default :* `true`
`Al`	MonStgElt	*Default :* “*LaScala*”

Given an R-module M, return a free resolution M as a complex C, and a comparison homomorphism $f : C_0 \to M$ (where C_0 is the term of C of degree 0).

By default, the free resolution will be **minimal**. Setting the parameter `Minimal` to `false` will construct a non-minimal resolution (which is constructed via a sequence of successive syzygy modules, with no minimization).

Magma has two algorithms for computing resolutions:

(1) The **La Scala** (LS) [SS98] algorithm, which works with a homogeneous module. The Magma implementation involves an extension of this algorithm which uses techniques from the Faugère F_4 [Fau99] algorithm to compute many normal forms together in a block.

(2) The **Iterative** algorithm, which simply computes successive syzygy modules progressively (minimizing as it goes if and only a minimal resolution is desired).

By default the LS algorithm is used if M is homogeneous and the coefficient ring of R is a finite field or the rational field, since this tends to be faster in general. But for some inputs the iterative algorithm may be significantly faster, particularly for some modules over the rationals. So one may set the parameter `Al` to `"Iterative"` to select the iterative algorithm. Uniqueness of the terms in the resolution is as follows.

(1) If M is homogeneous or defined over a local ring R, then the resulting complex C is guaranteed to be minimal, so the ranks of the terms in C and the associated Betti numbers will be unique.

(2) If M is non-homogeneous and over a global ring R, then the boundary maps of C will not have any entries which are units, but C cannot be guaranteed to be an absolutely minimal free resolution, so the ranks of the terms and the associated Betti numbers will not be unique in general. Also, Magma may choose to compute C by computing the free resolution C_H of a homogenization M_H of M, and then specializing C_H to yield C, since this method is usually faster (since the LS algorithm can then be used). One may set the parameter `Homogenize` to `true` or `false` to force Magma to use this homogenization technique or not.

If the parameter `Limit` is set to a non-zero value l, then at most l terms (plus the term corresponding to the free module) are computed. If R is an affine algebra or exterior algebra of rank n, then by default the limit is set to n, since the resolution is not finite in general.

```
SetVerbose("Resolution", v)
```

(Procedure.) Change the verbose printing level for the free resolution algorithm and related functions to be v.

Example H109E7

We construct the module $M = R^1/I$ where I is the ideal of the twisted cubic and then construct a minimal free resolution of M and note simple properties of this.

```
> R<x,y,z,t> := PolynomialRing(RationalField(), 4, "grevlex");
> B := [
>     -x^2 + y*t, -y*z + x*t, x*z - t^2,
>     x*y - t^2, -y*z + x*t, -x^2 + z*t
> ];
> M := GradedModule(Ideal(B));
> M;
Graded Module R^1/<relations>
Relations:
[-x^2 + y*t],
[-y*z + x*t],
[ x*z - t^2],
[ x*y - t^2],
[-y*z + x*t],
[-x^2 + z*t]
> C := FreeResolution(M);
> C;
Chain complex with terms of degree 4 down to -1
Dimensions of terms: 0 1 5 5 1 0
> Terms(C);
[
    Free Graded Module R^0,
    Free Graded Module R^1 with grading [5],
    Free Graded Module R^5 with grading [3, 3, 3, 3, 3],
    Free Graded Module R^5 with grading [2, 2, 2, 2, 2],
    Free Graded Module R^1,
    Free Graded Module R^0
]
> B := BoundaryMaps(C);
> B;
[*
    Graded module homomorphism (0 by 1),
    Graded module homomorphism (1 by 5) of degree 0
    Ambient matrix:
    [ x*z - t^2  x^2 - z*t -y*t + z*t  y*z - x*t -x*y + t^2],
    Graded module homomorphism (5 by 5) of degree 0
    Ambient matrix:
    [-y  x  0 -t  0]
    [ 0 -z  y  0  t]
    [ t -z  0  x  0]
    [ 0 -t  t  0  x]
    [-z  0  x -t  z],
    Graded module homomorphism (5 by 1) of degree 0
    Ambient matrix:
    [x^2 - z*t]
    [x*y - t^2]
    [x*z - t^2]
    [y*z - x*t]
    [y*t - z*t],
    Graded module homomorphism (1 by 0)
*]
> B[2]*B[3];
Module homomorphism (1 by 5)
Ambient matrix:
```

```
[0 0 0 0 0]
> B[3]*B[4];
Module homomorphism (5 by 1)
Ambient matrix:
[0]
[0]
[0]
[0]
[0]
> Image(B[3]) eq Kernel(B[4]);
true
```

Example H109E8

Following [CLO98, p.248], we compute the ideal I of $\mathbf{Q}[x,y]$ whose affine variety is a certain list of 6 pairs.

```
> R<x,y> := PolynomialRing(RationalField(), 2, "grevlex");
> L := [<0, 0>, <1, 0>, <0, 1>, <2, 1>, <1, 2>, <3, 3>];
> I := Ideal(L, R);
> I;
Ideal of Polynomial ring of rank 2 over Rational Field
Graded Reverse Lexicographical Order
Variables: x, y
Inhomogeneous, Dimension 0
Groebner basis:
[
    x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y,
    x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y,
    x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y,
    y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y
]
```

I is not homogeneous, and we compute a non-minimal free resolution of the module R/I.

```
> M := QuotientModule(I);
> M;
Reduced Module R^1/<relations>
Relations:
[  x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y],
[x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y],
[x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y],
[  y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y]
> C := FreeResolution(M: Minimal := false);
> C;
Chain complex with terms of degree 3 down to -1
Dimensions of terms: 0 3 4 1 0
> B := BoundaryMaps(C);
> B;
```

```
[*
    Module homomorphism (0 by 3),
    Module homomorphism (3 by 4)
    Ambient matrix:
    [-y + 5  x - 8      6     -2]
    [     4 -y - 8  x + 8     -4]
    [     2     -6 -y + 8  x - 5],
    Module homomorphism (4 by 1)
    Ambient matrix:
    [  x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y]
    [x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y]
    [x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y]
    [  y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y],
    Module homomorphism (1 by 0)
*]
> IsZero(B[2]*B[3]);
true
```

As noted in [CLO98], the 3 by 3 minors of the boundary map from R^3 to R^4 generate the ideal I again, and this is due to the Hilbert-Burch Theorem.

```
> U := Minors(Matrix(B[2]), 3);
> U;
[
    y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y,
    x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y,
    x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y,
    x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y
]
> Ideal(U) eq I;
true
```

109.12.2 Betti Numbers and Related Invariants

Each of the functions in this section compute numerical properties of a free resolution of a module M. Each function takes the same parameters as the function FreeResolution (not repeated here), thus allowing control of the construction of the underlying resolution.

In particular, by default the **minimal** free resolution of M is used (so the Betti numbers correspond to that), so the relevant invariant is guaranteed to be unique if M is graded or over a local ring R. Otherwise, one may set the parameter Minimal to false to give the Betti numbers for a non-minimal resolution.

Note: If M is graded and the LS algorithm is used (which will be the case by default), then computing any of the invariants to do with Betti numbers in this section may be quicker than computing the full resolution (since minimization of the actual resolution is needed for the latter). Thus it is preferable just to use one of the following functions instead of FreeResolution if only the numerical invariants are desired.

```
BettiNumbers(M)
```

Given a module M, return the Betti numbers of M, which is simply the sequence of integers consisting of the degrees of the non-zero terms of the free resolution of M. See the discussion above concerning the parameters. Since the underlying resolution is minimal by default, if M is graded or over a local ring, then the result is unique.

```
BettiNumber(M, i, j)
```

Given a module M and integers $i, j \geq 0$, return the graded Betti number $\beta_{i,j}$ of M as an integer. This is the number of generators of degree j in the i-th term F_i of the free resolution of M.

```
MaximumBettiDegree(M, i)
```

Given a module M and an integer $i \geq 0$, return the maximum degree of the generators in the i-th term of the free resolution of M. Equivalently, this is the maximum j such that `BettiNumber(M, i, j)` is non-zero.

```
BettiTable(M)
```

Given a module M, return the Betti table of M as a sequence S of sequences of integers, and a shift s. This is designed so that if M is non-zero, then `S[1, 1]` is always non-zero and `S[i, j]` equals `BettiNumber(M, i, j - i + s)`. (So the degrees are shifted by s.)

```
Regularity(M)
```

Given an R-module M which is either graded or over a local ring, return the Castelnuovo-Mumford regularity. This is the least r such that in a minimal free resolution of M, the maximum of the degrees of the generators of the i-th term F_i is at most $i + r$. A simple consequence of this is that M is generated by elements of degree at most r. See [Eis95, Sec. 20.5] or [DL06, p. 167].

```
HomologicalDimension(M)
```

Given a module M, return the homological dimension of M. This is just the length of a minimal free resolution of M (the number of non-zero boundary maps).

Example H109E9

For an integer n, we can construct a Koszul complex as the free resolution of R/I, where I is the ideal of $R = K[x_1, \ldots, x_n]$ generated by the n variables.

```
> Q := RationalField();
> n := 3;
> R<[x]> := PolynomialRing(Q, n);
> I := Ideal([R.i: i in [1 .. n]]);
> M := QuotientModule(I);
> M;
Graded Module R^1/<relations>
Relations:
```

```
[x[1]],
[x[2]],
[x[3]]
> C := FreeResolution(M);
> C;
Chain complex with terms of degree 4 down to -1
Dimensions of terms: 0 1 3 3 1 0
> BoundaryMaps(C);
[*
    Module homomorphism (0 by 1),
    Module homomorphism (1 by 3)
    Ambient matrix:
    [ x[3] -x[2]  x[1]],
    Module homomorphism (3 by 3)
    Ambient matrix:
    [-x[2]  x[1]     0]
    [-x[3]     0  x[1]]
    [    0 -x[3]  x[2]],
    Module homomorphism (3 by 1)
    Ambient matrix:
    [x[1]]
    [x[2]]
    [x[3]],
    Module homomorphism (1 by 0)
*]
```

In general, the i-th Betti number is $\binom{n}{i}$. We can see this for $n = 10$. Each boundary map consists of linear relations alone, so the regularity is zero.

```
> n := 10;
> R<[x]> := PolynomialRing(Q, n);
> I := Ideal([R.i: i in [1 .. n]]);
> M := QuotientModule(I);
> time C := FreeResolution(M);
Time: 0.060
> C;
Chain complex with terms of degree 11 down to -1
Dimensions of terms: 0 1 10 45 120 210 252 210 120 45 10 1 0
> Terms(C);
[
    Free Graded Module R^0,
    Free Graded Module R^1 with grading [10],
    Free Graded Module R^10 with grading [9, 9, 9, 9, 9, 9, 9, 9, 9, 9],
    Free Graded Module R^45 with grading [8^^45],
    Free Graded Module R^120 with grading [7^^120],
    Free Graded Module R^210 with grading [6^^210],
    Free Graded Module R^252 with grading [5^^252],
    Free Graded Module R^210 with grading [4^^210],
    Free Graded Module R^120 with grading [3^^120],
    Free Graded Module R^45 with grading [2^^45],
    Free Graded Module R^10 with grading [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    Free Graded Module R^1,
    Free Graded Module R^0
]
> B := BoundaryMaps(C);
> B: Minimal;
[*
    Graded module homomorphism (0 by 1),
    Graded module homomorphism (1 by 10) of degree 0,
    Graded module homomorphism (10 by 45) of degree 0,
    Graded module homomorphism (45 by 120) of degree 0,
    Graded module homomorphism (120 by 210) of degree 0,
    Graded module homomorphism (210 by 252) of degree 0,
    Graded module homomorphism (252 by 210) of degree 0,
    Graded module homomorphism (210 by 120) of degree 0,
    Graded module homomorphism (120 by 45) of degree 0,
    Graded module homomorphism (45 by 10) of degree 0,
    Graded module homomorphism (10 by 1) of degree 0,
    Graded module homomorphism (1 by 0)
*]
> [Binomial(n, i): i in [0 .. n]];
[ 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1 ]
> BettiTable(M);
[
    [ 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1 ]
]
> $1 eq [[Binomial(n, i): i in [0 .. n]]];
true
> Regularity(M);
0
```

Example H109E10

We can construct the same type of ideal and module as in the last example for $n = 3$, but over an exterior algebra. The free resolution is infinite here, but we can construct the resolution partially (by default, a bound is set on the number of terms). In this general construction, the i-th Betti number will be $\binom{i}{n-1}$.

```
> Q := RationalField();
> n := 3;
> R<[x]> := ExteriorAlgebra(Q, n);
> I := Ideal([R.i: i in [1 .. n]]);
> M := QuotientModule(I);
> M;
Reduced Module R^1/<relations>
Relations:
[x[1]],
[x[2]],
[x[3]]
> BettiNumbers(M);
[ 1, 3, 6, 10, 15 ]
> [Binomial(i + n - 1, n - 1): i in [0..4]];
[ 1, 3, 6, 10, 15 ]
> C := FreeResolution(M);
> C;
Chain complex with terms of degree 5 down to -1
Dimensions of terms: 0 15 10 6 3 1 0
> BoundaryMaps(C);
[*
    Graded module homomorphism (0 by 15),
    Graded module homomorphism (15 by 10) of degree 0
    Ambient matrix:
    [x[3]    0    0    0    0    0    0    0    0    0]
    [   0 x[3]    0 x[2]    0    0    0    0    0    0]
    [   0    0 x[2]    0    0    0    0    0    0    0]
    [   0 x[2] x[3]    0    0    0    0    0    0    0]
    [x[2]    0    0 x[3]    0    0    0    0    0    0]
    [   0    0    0    0 x[3]    0    0    0    0 x[1]]
    [   0    0    0    0    0 x[2]    0    0 x[1]    0]
    [   0    0    0    0 x[2] x[3]    0 x[1]    0    0]
    [   0    0    0    0    0    0 x[1]    0    0    0]
    [   0    0    0    0    0 x[1] x[2]    0    0    0]
    [   0    0    0    0 x[1]    0 x[3]    0    0    0]
    [   0    0    0 x[1]    0    0    0 x[3]    0 x[2]]
    [   0    0 x[1]    0    0    0    0    0 x[2]    0]
    [   0 x[1]    0    0    0    0    0 x[2] x[3]    0]
    [x[1]    0    0    0    0    0    0    0    0 x[3]],
    Graded module homomorphism (10 by 6) of degree 0
    Ambient matrix:
    [x[3]    0    0    0    0    0]
    [   0 x[3] x[2]    0    0    0]
    [   0 x[2]    0    0    0    0]
    [x[2]    0 x[3]    0    0    0]
    [   0    0    0 x[3]    0 x[1]]
    [   0    0    0 x[2] x[1]    0]
    [   0    0    0 x[1]    0    0]
    [   0    0 x[1]    0 x[3] x[2]]
    [   0 x[1]    0    0 x[2]    0]
    [x[1]    0    0    0    0 x[3]],
    Graded module homomorphism (6 by 3) of degree 0
    Ambient matrix:
    [x[3]    0    0]
    [   0 x[2]    0]
    [x[2] x[3]    0]
    [   0    0 x[1]]
    [   0 x[1] x[2]]
    [x[1]    0 x[3]],
    Graded module homomorphism (3 by 1) of degree 0
    Ambient matrix:
    [x[3]]
    [x[2]]
    [x[1]],
    Graded module homomorphism (1 by 0)
*]
```

Example H109E11

We construct a non-homogeneous quotient module M of $\mathbf{Q}^3$. As expected, the Betti numbers of the localization of M are smaller than the Betti numbers of M.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3, "grevlex");
> R3 := RModule(R, 3);
> B := [R3 | [x*y, x^2, z], [x*z^3, x^3, y], [y*z, z, x],
>       [z, y*z, x], [y, z, x]];
> M := quo<R3 | B>;
> M;
Reduced Module R^3/<relations>
Relations:
[  x*y,   x^2,     z],
[x*z^3,   x^3,     y],
[  y*z,     z,     x],
[    z,   y*z,     x],
[    y,     z,     x]
> BettiNumbers(M);
[ 3, 5, 4, 2 ]
> BettiNumbers(Localization(M));
[ 3, 5, 3, 1 ]
```

Since M is non-homogeneous, the Betti numbers are not unique. If we create a second module M_2 which is equivalent to M and compute the Betti numbers this time without homogenization (in the internal free resolution algorithm), then we obtain different Betti numbers for M_2. But since the Betti numbers over a local ring are unique, we get the same result for the localization of M_2.

```
> M2 := quo<R3 | B>;
> BettiNumbers(M2: Homogenize :=false);
[ 3, 6, 5, 2 ]
> BettiNumbers(Localization(M2): Homogenize:=false);
[ 3, 5, 3, 1 ]
```

Example H109E12

Suppose M is a graded R-module. Given the graded Betti numbers $\beta_{i,j}$ of M, one can compute the Hilbert series $H_M(t)$ of M via the formula ([Eis95, Thm. 1.13] or [DL06, Thm. 1.22]):

$$H_M(t) = \frac{\sum_{i,j} (-1)^i \beta_{i,j} t^j}{D},$$

where D is the Hilbert denominator of M: this depends on the underlying ring R and equals

$$\prod_{i=1}^{n} (1 - t^{w_i}),$$

where n is the rank of R and w_i is the weight of the i-th variable (1 by default). We can thus write a simple function to compute the Hilbert series numerator via this formula.

```
> function HilbertNumeratorBetti(M)
>    P<t> := PolynomialRing(IntegerRing());
>    return &+[
>        (-1)^i*BettiNumber(M, i, j)*t^j:
>           j in [0 .. MaximumBettiDegree(M, i)],
>           i in [0 .. #BettiNumbers(M)]
>    ];
> end function;
```

We then check that this function agrees with the Magma internal function `HilbertNumerator` for some modules. (Since the modules do not have negative gradings, we do not have to worry about the denominator shift which is 0 for these modules.) First we try the Twisted Cubic.

```
> Q := RationalField();
> R<x,y,z,t> := PolynomialRing(Q, 4, "grevlex");
> B := [
>     -x^2 + y*t, -y*z + x*t, x*z - t^2,
>     x*y - t^2, -y*z + x*t, -x^2 + z*t
> ];
> M := GradedModule(Ideal(B));
> HilbertNumeratorBetti(M);
-t^5 + 5*t^3 - 5*t^2 + 1
> HilbertNumerator(M);
-t^5 + 5*t^3 - 5*t^2 + 1
0
```

Now we apply the function to the module $M = R^1/I$ where I is the ideal generated by the 2×2 minors of a generic 4×4 matrix. Computing the Hilbert series numerator via the Betti numbers takes a little time since the resolution is non-trivial. Note the components of the Betti table which contribute to the terms of the Hilbert series numerator.

```
> n := 4;
> R<[x]> := PolynomialRing(Q, n^2, "grevlex");
> A := Matrix(n, [R.i: i in [1 .. n^2]]);
> A;
[ x[1]  x[2]  x[3]  x[4]]
[ x[5]  x[6]  x[7]  x[8]]
[ x[9] x[10] x[11] x[12]]
[x[13] x[14] x[15] x[16]]
> I := Ideal(Minors(A, 2));
> #Basis(I);
36
> M := QuotientModule(I);
> time HilbertNumeratorBetti(M);
-t^12 + 36*t^10 - 160*t^9 + 315*t^8 - 288*t^7 + 288*t^5 - 315*t^4 + 160*t^3 -
    36*t^2 + 1
Time: 0.470
> time HilbertNumerator(M);
-t^12 + 36*t^10 - 160*t^9 + 315*t^8 - 288*t^7 + 288*t^5 - 315*t^4 + 160*t^3 -
    36*t^2 + 1
0
Time: 0.000
> assert $1 eq $2;
> BettiNumbers(M);
[ 1, 36, 160, 315, 388, 388, 315, 160, 36, 1 ]
> BettiTable(M);
[
    [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
    [ 0, 36, 160, 315, 288, 100, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 100, 288, 315, 160, 36, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
]
0
```

Example H109E13

Given a graded module $M = R^1/I$, one can obtain an upper bound on the regularity of M by computing the regularity of $M_L = R^1/I_L$, where I_L is the leading monomial ideal of I. This will be faster in general (since the associated free resolution will be easier to compute).

```
> wts := [ 1, 5, 9, 13, 17, 5, 1, 1, 1 ];
> K := GF(32003);
> R<x0,x1,x2,x3,x4,y0,y1,u,t> := PolynomialRing(K, wts);
> I := Ideal([
>     x0*y0 - y1^3*u^3 - x1*t,
>     x1*y1 - x0*u^5 - t^6,
>     x1^2 - x0*x2 + y1^2*u^3*t^5,
>     x2^2 - x1*x3 + y0*y1*u^8*t^4,
>     x3^2 - x2*x4 + y0^2*u^13*t^3,
>     x3*y0 - u^18 - x4*t,
>     x4*y1 - x3*u^5 - y0^3*t^3,
>     x1*x2 - x0*x3 + y0*y1^2*u^3*t^4 + y1*u^8*t^5,
>     x2^2 - x0*x4 + y0*y1*u^8*t^4 + u^13*t^5,
>     x2*x3 - x1*x4 + y0^2*y1*u^8*t^3 + y0*u^13*t^4,
>     x1*y0 - y1^2*u^8 - x2*t,
>     x2*y0 - y1*u^13 - x3*t,
>     x2*y1 - x1*u^5 - y0*t^5,
>     x3*y1 - x2*u^5 - y0^2*t^4]);
> IsHomogeneous(I);
true
> M := GradedModule(I);
> time Regularity(M);
67
Time: 3.360
> IL := LeadingMonomialIdeal(I);
> ML := GradedModule(IL);
> time Regularity(ML);
92
Time: 0.530
> BettiNumbers(M);
[ 1, 14, 45, 72, 76, 58, 29, 8, 1 ]
> BettiNumbers(ML);
[ 1, 42, 210, 505, 723, 659, 388, 144, 31, 3 ]
```

Example H109E14

The following example shows how to explicitly use the resolution and syzygy functions to compute the ideal of a random space curve (in P^3) of genus 11. The construction is described in Section 1.2 of [ST02] and an equivalent form of the following computation is used by Magma's RandomCurveByGenus function to produce such curves.

We work over the field $GF(101)$, which will be referred to as K and the polynomial ring R will be the 4 variable polynomial ring over K. The construction begins by choosing a random 8 x 3 matrix with entries given by random linear and quadratic polynomials of R in appropriate positions. The minimal free resolution of the reduced module having this as the matrix of relations is computed. The image of the second boundary map of the resolution is the module referred to as $\mathcal{G}^*$ in the above reference. Taking the submatrix of rows of a certain weighting of the matrix defining this map, we multiply by a 6 x 8 matrix with random entries in K. The resulting matrix represents a map from a free module F of rank 6 to $\mathcal{G}^*$, whose kernel is isomorphic to R as a submodule of F. The 6 coordinates of a generator of the kernel generate the desired ideal I. This kernel is computed with a syzygy computation (note: we could also use Kernel for the matrix giving the map). We also check that the quotient module of I has a minimal free resolution of the right form.

```
> K := GF(101);
> R<x,y,z,t> := PolynomialRing(GF(101),4,"grevlex");
> v := [1,1,1,1,1,1,2,2];
> // generate the base random relations with appropriate linear and quadratic
> // entries using the Random function for multivariate polynomials.
> rels := [[Random(i,R,0): j in [1..3]] : i in v];
> Matrix(8,3,[TotalDegree(e) : e in &cat(rels)]);
```

```
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
[2 2 2]
[2 2 2]
> // get the quotient module
> F := RModule(R,3);
> M := quo<F|rels>;
```

Get the minimal free resolution and check that it has the correct Betti table.

```
> res := MinimalFreeResolution(M);
> BettiTable(res);
[
    [ 3, 6, 0, 0, 0 ],
    [ 0, 2, 8, 0, 0 ],
    [ 0, 0, 3, 10, 4 ]
]
0
```

Get the 2nd boundary map matrix and then the 8 x 8 submatrix of linear and quadratic entry rows.

```
> mat :=  Matrix(BoundaryMap(res,2));
> Nrows(mat); Ncols(mat);
11
8
> u := [1,1,2,2,2,2,2,2];
> mat := Matrix(R,[ri : i in [1..11] |
>          &and[(ri[j] eq 0) or (TotalDegree(ri[j]) eq u[j]):
>             j in [1..8]] where ri is Eltseq(mat[i])]);
> Nrows(mat); Ncols(mat);
8
8
> Matrix(8,8,[TotalDegree(m) :m in Eltseq(mat)]);
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
```

Now generate the random 6 x 8 matrix over K, compute the kernel of the composition via syzygies and generate the matrix I.

```
> mat1 := Matrix(R,6,8,[Random(K) : i in [1..48]]);
> matc := mat1*mat;
> F1 := EModule(R,[2-x : x in u]);
> syz := SyzygyModule(sub<F1|RowSequence(matc)>);
> B := MinimalBasis(syz);
> #B;
1
> I := ideal<R|Eltseq(B[1])>;
```

Finally, check I has the right dimension (2) and degree (12) and that R/I has the correct minimal free resolution with Betti table as given in [ST02].

```
> Dimension(I); Degree(I);
2 [ 3, 4 ]
12
> OC := QuotientModule(I);
> BettiTable(MinimalFreeResolution(OC));
[
    [ 1, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 6, 2, 0 ],
    [ 0, 0, 6, 3 ]
]
0
```

109.13 The Hom Module and Ext

`Hom(M, N)`

Given $R-$modules M and N, return $H = \mathrm{Hom}_R(M, N)$ as an abstract reduced module and a transfer map $f : H \to S$, where S is the set of all homomorphisms (of type ModMPolHom) from M to N.

Thus H is a module representing the set of all homomorphisms from M to N, while f maps an element $h \in H$ to an actual homomorphism from M to N (and the inverse image of an element of S under f gives a corresponding element of H).

If M and N are graded, then H is graded also, and the degree d_f of an element $f \in H$ is the degree of the corresponding homomorphism (so an element in M of degree d will be mapped by f to zero or an element of degree $d_f + d$ in N).

`Hom(C, N)`

Given a complex C of R-modules and an R-module N, return $\mathrm{Hom}_R(C, N)$. This is a new complex whose i-th term is $\mathrm{Hom}_R(C_i, N)$ (where C_i is the i-th term of C); the boundary maps are also derived from those of C in the natural way via the functor $\mathrm{Hom}_R(-, N)$ (see [Eis95, p.63]). Note that the direction of arrows in this complex is opposite to that of C.

```
Ext(i, M, N)
```

Given an integer $i \geq 0$ and R-modules M and N, return $\mathrm{Ext}^i(M, N)$. This is the homology at the i-th term of the complex $\mathrm{Hom}_R(C, N)$ where C is a free resolution of M.

Example H109E15

We construct a Hom module and explicit homomorphisms derived from it.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> M := quo<GradedModule(R, 3) |
>     [x*y, x*z, y*z], [y, x, y],
>     [0, x^3 - x^2*z, x^2*y - x*y*z], [y*z, x^2, x*y]>;
> N := quo<GradedModule(R, 2) |
>     [x^2, y^2], [x^2, y*z], [x^2*z, x*y^2]>;
> M;
Graded Module R^3/<relations>
Relations:
[         x*y,           x*z,           y*z],
[           y,             x,             y],
[           0,   x^3 - x^2*z, x^2*y - x*y*z],
[         y*z,           x^2,           x*y]
> N;
Graded Module R^2/<relations>
Relations:
[  x^2,   y^2],
[  x^2,   y*z],
[x^2*z, x*y^2]
> H, f := Hom(M, N);
> H;
Graded Module R^7/<relations> with grading [1, 2, 1, 1, 1, 1, 1]
Relations:
[x, 0, 0, -z, 0, x, 0],
[y, 0, x, 0, y, 0, 0],
[y, 0, x, 0, 0, y, 0],
[0, 0, 0, 0, 0, 0, y],
[-y, 0, -x, 0, -z, 0, z],
[x, 0, 0, -y, x, 0, 0],
[x*y, y, 0, 0, 0, 0, x*y],
[-x*y + x*z, -y + z, 0, 0, 0, 0, x*z - z^2],
[x*z, x, 0, 0, 0, 0, 0],
[0, y, 0, y^2, -z^2, 0, z^2],
[0, y - z, 0, 0, 0, 0, z^2]
> h := f(H.1);
> h;
Module homomorphism (3 by 2) of degree 1
Presentation matrix:
[0 z]
```

```
[x 0]
[0 0]
> $1 @@ f;
[1, 0, 0, 0, 0, 0, 0]
> Degree(M.1);
0
> h(M.1);
[0, z]
> Degree(h(M.1));
1
> f(Basis(H));
[
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [0 z]
    [x 0]
    [0 0],
    Module homomorphism (3 by 2) of degree 2
    Presentation matrix:
    [   0 -z^2]
    [   0  y*z]
    [   0    0],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0  0]
    [-y  0]
    [ x  0],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0  0]
    [ 0 -y]
    [ 0  x],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0 -z]
    [ 0  0]
    [ 0  y],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0 -z]
    [ 0  0]
    [ 0  z],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [    0 y - z]
    [    0     0]
    [    0     0]
]
```

109.14 Tensor Products and Tor

`TensorProduct(M, N)`

Given $R-$modules M and N, return the tensor product $M \otimes_R N$ as an ambient module T, together with the associated map $f : M \times N \to T$. If M and N are graded, then T is graded also.

`TensorProduct(C, N)`

Given a complex C of R-modules and an R-module N, return $C \otimes_R N$. This is a new complex whose i-th term is $C_i \otimes_R N$ (where C_i is the i-th term of C); the boundary maps are also derived from those of C in the natural way via the functor $- \otimes_R N$ (see [Eis95, p.64]).

`Tor(i, M, N)`

Given an integer $i \geq 0$ and R-modules M and N, return $\mathrm{Tor}^i(M, N)$. This is the homology at the i-th term of the complex $C \otimes_R N$ where C is a free resolution of M.

Example H109E16

We construct a tensor product and some Tor modules for the same modules from the previous example.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> M := quo<GradedModule(R, 3) |
>     [x*y, x*z, y*z], [y, x, y],
>     [0, x^3 - x^2*z, x^2*y - x*y*z], [y*z, x^2, x*y]>;
> N := quo<GradedModule(R, 2) |
>     [x^2, y^2], [x^2, y*z], [x^2*z, x*y^2]>;
> T, f := TensorProduct(M, N);
> T;
Graded Module R^6/<relations>
Relations (Groebner basis):
[x^2, y*z, 0, 0, 0, 0],
[0, 0, 0, 0, x^2, y*z],
[0, 0, 0, 0, 0, x*y*z - y*z^2],
[x*y - y*z, 0, 0, 0, 0, 0],
[0, x*y - y*z, 0, 0, 0, 0],
[y*z, 0, 0, -y*z, x*y, 0],
[y, 0, x, 0, y, 0],
[0, y, 0, x, 0, y],
[0, y^2 - y*z, 0, 0, 0, 0],
[0, 0, 0, y^2 - y*z, 0, 0],
[0, 0, 0, 0, 0, y^2 - y*z],
[y*z^2, 0, 0, -y*z^2, 0, -y*z^2],
[0, y*z^2, 0, y*z^2, 0, y*z^2]
```

Note that f maps the cartesian product of M and N into T.

```
> f(<M.1, N.1>);
[1, 0, 0, 0, 0, 0]
> [f(<m, n>): n in Basis(N), m in Basis(M)];
[
    [1, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1]
]
```

Finally we construct associated Tor modules.

```
> Tor(0, M, N);
Graded Module R^6/<relations>
Relations:
[y, 0, x, 0, y, 0],
[0, y, 0, x, 0, y],
[0, 0, 0, 0, x*y - y*z, 0],
[0, 0, 0, 0, 0, x*y - y*z],
[y*z, x^2, 0, 0, 0, 0],
[x*y*z - y*z^2, 0, 0, 0, 0, 0],
[y^2 - y*z, 0, 0, 0, 0, 0],
[0, 0, y*z, x^2, 0, 0],
[0, 0, x*y*z - y*z^2, 0, 0, 0],
[0, 0, y^2 - y*z, 0, 0, 0],
[0, 0, 0, 0, y*z, x^2],
[0, 0, 0, 0, y^2 - y*z, 0],
[0, 0, 0, 0, x*y*z - y*z^2, 0]
> Tor(1, M, N);
Graded Module R^2/<relations> with grading [3, 3]
Relations:
[y - z,     0],
[    z,    -y],
[  z^2,  -x*y],
[    0,     0]
> Tor(2, M, N);
Free Reduced Module R^0
```

109.15 Cohomology Of Coherent Sheaves

We have implemented functions to compute the dimensions of cohomology groups of coherent sheaves on ordinary projective space over an (exact) field. The sheaves are represented by graded modules over the coordinate ring of the ambient projective space. The sheaf may arise naturally as one supported on a particular closed subscheme (*eg*, the structure sheaf of a projective variety) but it is a matter of indifference whether the sheaf is considered as lying on the subscheme or the entire ambient space (equivalently, whether the representing module is considered as a module over the coordinate ring of the ambient or the quotient coordinate ring of the subscheme) because the cohomology groups are naturally isomorphic. We plan to add a fuller package of functionality for coherent sheaves, but it is convenient to add the cohomology function now as the algorithm we use works equally efficiently (roughly speaking) when applied to any two graded modules that represent the same coherent sheaf.

The algorithm we have implemented is that of Decker, Eisenbud, Floystad and Schreyer which uses the Beilinson-Gelfand-Gelfand (BGG) correspondence to reduce the computation of the cohomology groups of the sheaf and its Serre twists to that of various graded free modules in the projective resolution of a module over a finite exterior (alternating) algebra.

```
CohomologyDimension(M,r,n)
```

Verbose `Cohom` *Maximum* : 1

M is a graded module over $P = k[x_0, .., x_m]$ with k an exact field. Let $\tilde{M}$ be the corresponding coherent sheaf on $Proj(P) = \mathbf{P}^m_k$. The function returns the k-dimension of the cohomology group $H^r(\mathbf{P}^m_k, \tilde{M}(n))$ where $\tilde{M}(n)$ is the nth Serre twist of $\tilde{M}$. n can be any integer and r a non-negative integer.

The algorithm used is based on the BGG correspondence. Details can be found in [EFS03] or see [DE02] for a slightly more computational description. Let A be the finite exterior algebra with $m+1$ generators, which is of dimension 2^{m+1} over k. The *Tate resolution* of $\tilde{M}$ is a doubly infinite exact sequence of graded free A-modules. Each cohomology group of a twist of $\tilde{M}$ is isomorphic as a k vector space to a particular graded piece of a particular term in the Tate resolution. In fact, we never need to explicitly compute the terms of the resolution of index $\geq reg(M)$ (the regularity of M) because they are pure graded of dimension given by the Hilbert polynomial of M.

The algorithm computes two consecutive terms in the Tate resolution at indices $\geq reg(M)$, and the A-homomorphism between them, from two corresponding graded pieces of M and the linear maps between them coming from multiplication by the base variables. Then the resolution is extended backwards as far as necessary by computing the A-projective resolution of the kernel of this A-homomorphism. The projective resolution is efficiently determined by non-commutative Gröbner basis computations. This uses the new Magma machinery for exterior algebras and their modules. The projective resolution information is cached so that repeated calls to the function for the same module M will require either no extra work or only an extension to the part of the resolution already computed.

Example H109E17

We consider a random surface X in a family of Enriques surfaces of degree 9 in $\mathbf{P}^4$. It is defined by 15 degree 5 polynomials and we work over $\mathbf{F}_{17}$ to keep the input of reasonable size (it is still fairly large!).
The surface is non-singular with arithmetic genus (p_a), geometric genus (g) and irregularity q all zero. These are related generally for a non-singular surface by $g = p_a + q$ and p_a can be computed without cohomology machinery (from Hilbert polynomials). But cohomology of the structure sheaf of X and Serre duality is the easiest way to get g or q.

```
> R<x,y,z,t,u> := PolynomialRing(GF(17),5,"grevlex");
> I := ideal<R |[
>  2*x^3*z*t + 5*x^2*y*z*t + 14*x^2*z^2*t + x^3*z*u + 5*x^2*y*z*u +
>   10*x^2*z^2*u + 8*x*y*z^2*u + 15*y^2*z^2*u + 4*x*z^3*u + 2*y*z^3*u +
>   9*x^3*t*u + 14*x^2*y*t*u + 16*x^2*z*t*u + 10*x*y*z*t*u + 10*x*z^2*t*u +
>   y*z^2*t*u + 13*x^3*u^2 + 14*x^2*y*u^2 + 11*x^2*z*u^2 + 15*x*y*z*u^2 +
>   10*y^2*z*u^2 + 8*x*z^2*u^2 + 8*y*z^2*u^2 + x^2*t*u^2 + 11*x*y*t*u^2 +
>   16*x*y*u^3 + 9*y^2*u^3 + 4*x*z*u^3 + 2*y*z*u^3 + 10*x*t*u^3 + y*t*u^3 +
>   8*x*u^4 + 8*y*u^4,
>  5*x^3*z*t + x^2*z^2*t + 5*x^3*z*u + 11*x^2*z^2*u + 15*x*y*z^2*u + 2*x*z^3*u
>   + 14*x^3*t*u + 5*x^2*z*t*u + x*z^2*t*u + 14*x^3*u^2 + 10*x*y*z*u^2 +
>   8*x*z^2*u^2 + 15*x^2*t*u^2 + 11*x^2*u^3 + 9*x*y*u^3 + 2*x*z*u^3 +
>   x*t*u^3 + 8*x*u^4,
>  14*x^3*z*t + x^2*y*z*t + 13*x^2*z^2*t + 7*x^2*z*t^2 + 3*x^3*z*u +
>   16*x^2*y*z*u + 4*x^2*z^2*u + 6*x^3*t*u + 16*x^2*y*t*u + 9*x^2*z*t*u +
>   9*x^2*t^2*u + 11*x^3*u^2 + x^2*y*u^2 + 14*x^2*z*u^2 + 2*x*z^2*u^2 +
>   11*y*z^2*u^2 + 6*z^3*u^2 + 4*x^2*t*u^2 + 4*x*z*t*u^2 + 14*y*z*t*u^2 +
>   6*z^2*t*u^2 + 15*x*t^2*u^2 + 10*z*t^2*u^2 + 3*x^2*u^3 + 11*x*z*u^3 +
>   16*y*z*u^3 + 4*z^2*u^3 + 16*x*t*u^3 + 3*y*t*u^3 + 14*z*t*u^3 + 6*t^2*u^3
>   + 13*x*u^4 + 7*y*u^4 + 16*z*u^4 + 11*t*u^4 + 10*u^5,
>  15*x^3*z^2 + 12*x^2*y*z^2 + 3*x^2*z^3 + 12*x^3*z*u + 8*x^2*y*z*u +
>   11*x^2*z^2*u + x^3*u^2 + 14*x^2*y*u^2 + 3*x^2*z*u^2 + 11*x^2*u^3,
>  12*x^3*z^2 + 16*x^2*z^3 + 8*x^3*z*u + x^2*z^2*u + 14*x^3*u^2 + 16*x^2*z*u^2
>   + x^2*u^3,
>  2*x^3*y*z + 5*x^2*y^2*z + 14*x^2*y*z^2 + 13*x^3*y*u + 12*x^2*y^2*u +
>   8*x^3*z*u + 4*x^2*y*z*u + 12*x^2*z^2*u + 3*x*y*z^2*u + 14*y^2*z^2*u +
>   15*x*z^3*u + 3*y*z^3*u + 15*x^2*y*t*u + 4*x^2*z*t*u + 15*x*y*z*t*u +
>   11*x*z^2*t*u + 10*y*z^2*t*u + 2*x^3*u^2 + 12*x^2*y*u^2 + 3*x^2*z*u^2 +
>   14*x*y*z*u^2 + 13*x*z^2*u^2 + 10*y*z^2*u^2 + x^2*t*u^2 + 15*x*y*t*u^2 +
>   16*y*z*t*u^2 + 12*x*y*u^3 + 3*y^2*u^3 + 15*x*z*u^3 + 4*y*z*u^3 +
>   11*x*t*u^3 + 6*y*t*u^3 + 13*x*u^4 + 14*y*u^4,
>  5*x^3*y*z + x^2*y*z^2 + 10*x^2*z^2*t + 4*x^4*u + 12*x^3*y*u + 5*x^3*z*u +
>   12*x^2*y*z*u + 14*x*y*z^2*u + 3*x*z^3*u + 15*x^3*t*u + 16*x^2*z*t*u +
>   10*x*z^2*t*u + 11*x^3*u^2 + 4*x^2*y*u^2 + 13*x^2*z*u^2 + 10*x*z^2*u^2 +
>   4*x^2*t*u^2 + 16*x*z*t*u^2 + 13*x^2*u^3 + 3*x*y*u^3 + 4*x*z*u^3 +
>   6*x*t*u^3 + 14*x*u^4,
>  10*x^2*z^3 + 8*x^2*z^2*u + 5*x^2*z*u^2 + 11*x^2*u^3,
>  16*x^3*z^2 + 12*x^2*y*z^2 + 7*x^2*z^3 + 9*x*y*z^3 + 2*y^2*z^3 + 13*x*z^4 +
>   15*y*z^4 + 13*x^3*z*t + 12*x^2*y*z*t + 7*x^2*z^2*t + 7*x*y*z^2*t +
>   7*x*z^3*t + 16*y*z^3*t + 4*x^3*z*u + 3*x^2*y*z*u + 6*x^2*z^2*u +
>   2*x*y*z^2*u + 7*y^2*z^2*u + 9*x*z^3*u + 9*y*z^3*u + 16*x^3*t*u +
>   3*x^2*y*t*u + 13*x^2*z*t*u + 6*x*y*z*t*u + x*y*z*u^2 + 8*y^2*z*u^2 +
>   13*x*z^2*u^2 + 15*y*z^2*u^2 + 6*x^2*t*u^2 + 7*x*z*t*u^2 + 16*y*z*t*u^2 +
>   9*x*z*u^3 + 9*y*z*u^3,
>  12*x^3*z^2 + 6*x^2*z^3 + 2*x*y*z^3 + 15*x*z^4 + 12*x^3*z*t + 11*x^2*z^2*t +
>   16*x*z^3*t + 3*x^3*z*u + 7*x*y*z^2*u + 9*x*z^3*u + 3*x^3*t*u +
>   3*x^2*z*t*u + 6*x^2*z*u^2 + 8*x*y*z*u^2 + 15*x*z^2*u^2 + 16*x^2*t*u^2 +
>   16*x*z*t*u^2 + 9*x*z*u^3,
>  x^3*y*z + 5*x^2*y^2*z + 10*x^2*y*z^2 + 8*x*y^2*z^2 + 15*y^3*z^2 + 4*x*y*z^3
>   + 2*y^2*z^3 + 13*x^3*y*t + 2*x^2*y^2*t + 9*x^3*z*t + 12*x^2*y*z*t +
>   10*x*y^2*z*t + 5*x^2*z^2*t + 7*x*y*z^2*t + 4*y^2*z^2*t + 2*x*z^3*t +
>   14*y*z^3*t + 2*x^2*y*t^2 + 13*x^2*z*t^2 + 2*x*y*z*t^2 + 6*x*z^2*t^2 +
>   7*y*z^2*t^2 + 13*x^3*y*u + 14*x^2*y^2*u + 11*x^2*y*z*u + 15*x*y^2*z*u +
>   10*y^3*z*u + 8*x*y*z^2*u + 8*y^2*z^2*u + 15*x^3*t*u + 6*x^2*y*t*u +
>   11*x*y^2*t*u + 14*x^2*z*t*u + 3*x*y*z*t*u + 4*x*z^2*t*u + 7*y*z^2*t*u +
>   16*x^2*t^2*u + 2*x*y*t^2*u + y*z*t^2*u + 16*x*y^2*u^2 + 9*y^3*u^2 +
>   4*x*y*z*u^2 + 2*y^2*z*u^2 + 15*x*y*t*u^2 + 15*y^2*t*u^2 + 2*x*z*t*u^2 +
>   13*y*z*t*u^2 + 6*x*t^2*u^2 + 11*y*t^2*u^2 + 8*x*y*u^3 + 8*y^2*u^3 +
>   4*x*t*u^3 + 3*y*t*u^3,
>  5*x^3*y*z + 11*x^2*y*z^2 + 15*x*y^2*z^2 + 2*x*y*z^3 + 13*x^4*t + 2*x^3*y*t +
>   7*x^3*z*t + 6*x^2*y*z*t + 16*x^2*z^2*t + 4*x*y*z^2*t + 14*x*z^3*t +
>   2*x^3*t^2 + 9*x^2*z*t^2 + 7*x*z^2*t^2 + 14*x^3*y*u + 5*x^3*z*u +
>   4*x^2*y*z*u + 10*x*y^2*z*u + 12*x^2*z^2*u + 8*x*y*z^2*u + 15*x*z^3*u +
>   6*y*z^3*u + 11*z^4*u + 16*x^3*t*u + 15*x^2*y*t*u + 13*x^2*z*t*u +
>   3*x*z^2*t*u + 3*y*z^2*t*u + 11*z^3*t*u + 13*x^2*t^2*u + 3*x*z*t^2*u +
>   7*z^2*t^2*u + 7*x^3*u^2 + 7*x^2*y*u^2 + 9*x*y^2*u^2 + x^2*z*u^2 +
>   2*x*y*z*u^2 + 6*x*z^2*u^2 + y*z^2*u^2 + 13*z^3*u^2 + 12*x^2*t*u^2 +
>   15*x*y*t*u^2 + 14*x*z*t*u^2 + 14*y*z*t*u^2 + 3*z^2*t*u^2 + 11*x*t^2*u^2
>   + 11*z*t^2*u^2 + 9*x^2*u^3 + 8*x*y*u^3 + 4*x*z*u^3 + 10*y*z*u^3 +
>   z^2*u^3 + 3*x*t*u^3 + 6*z*t*u^3 + 7*z*u^4,
>  4*x^4*z + 3*x^3*y*z + 10*x^3*z^2 + x^2*z^3 + 14*x*y*z^3 + 3*x*z^4 +
>   15*x^3*z*t + 8*x^2*z^2*t + 10*x*z^3*t + 14*x^3*y*u + 15*x^3*z*u +
>   11*x^2*z^2*u + 10*x*z^3*u + 16*x^2*z*t*u + 16*x*z^2*t*u + 6*x^3*u^2 +
>   14*x^2*z*u^2 + 3*x*y*z*u^2 + 4*x*z^2*u^2 + 6*x^2*t*u^2 + 6*x*z*t*u^2 +
>   15*x^2*u^3 + 14*x*z*u^3,
>  5*x^3*z^2 + 4*x^2*y*z^2 + 12*x^2*z^3 + 15*x*z^4 + 6*y*z^4 + 11*z^5 +
>   14*x^3*z*t + x^2*y*z*t + 7*x^2*z^2*t + 13*x*z^3*t + 3*y*z^3*t + 11*z^4*t
>   + 12*x^2*z*t^2 + 2*x*z^2*t^2 + 7*z^3*t^2 + 7*x^3*z*u + 13*x^2*y*z*u +
>   x^2*z^2*u + 6*x*z^3*u + y*z^3*u + 13*z^4*u + 6*x^3*t*u + 16*x^2*y*t*u +
>   9*x^2*z*t*u + x*z^2*t*u + 14*y*z^2*t*u + 3*z^3*t*u + 6*x^2*t^2*u +
>   11*z^2*t^2*u + 9*x^2*z*u^2 + 4*x*z^2*u^2 + 10*y*z^2*u^2 + z^3*u^2 +
>   15*x^2*t*u^2 + 6*z^2*t*u^2 + 7*z^2*u^3,
>  13*x^5 + 2*x^4*y + 7*x^4*z + 6*x^3*y*z + 16*x^3*z^2 + 4*x^2*y*z^2 +
>   14*x^2*z^3 + 2*x^4*t + 9*x^3*z*t + 7*x^2*z^2*t + 16*x^4*u + 15*x^3*y*u +
>   x^3*z*u + 11*x^2*z^2*u + 3*x*y*z^2*u + 14*x*z^3*u + 13*x^3*t*u +
>   3*x^2*z*t*u + 7*x*z^2*t*u + 14*x^3*u^2 + 15*x^2*y*u^2 + 6*x^2*z*u^2 +
>   14*x*y*z*u^2 + 10*x*z^2*u^2 + 11*x^2*t*u^2 + 11*x*z*t*u^2 + 3*x*z*u^3]>;
> X := Scheme(Proj(R),I); // define the surface
> // The structure sheaf O_X of X is represented by graded module R/I
> O_X := GradedModule(I); // R/I
```

We first compute the dimension of $H^0(O_X)$. This is fairly uninteresting (it's just 1) but after this computation, the cached data will allow the following cohomology calls to execute practically instantaneously.

```
> CohomologyDimension(O_X,0,0);  // dim H^0(O_X)
1
> // get the geometric genus and irregularity
> time CohomologyDimension(O_X,2,0); // dim H^2(O_X) = g
0
Time: 0.000
> time CohomologyDimension(O_X,1,0); // dim H^1(O_X) = q
0
Time: 0.000
> // => p_a(X)=0. Verify this.
> ArithmeticGenus(X);
0
```

We now compute a module representative of the canonical sheaf K_X of X. We can just take $Ext^2_R(M, R(-5))$. Then we check again that $H^0(K_X) = g$ is 0. Note that here the module representing K_X is maximal so that $H^0(K_X(n))$ is just the dimension of its nth graded part for any n. However, in other cases (where X is again *not* arithmetically Cohen-Macaulay) the Ext computation for K_X may not give the maximal representing module, so its 0th graded piece might have dimension less than g.

```
> K_X := Ext(2,O_X,RModule(R,[5]));
> K_X;
Reduced Module R^6/<relations> with grading [1, 1, 1, 1, 1, 1]
Quotient Relations:
[12*z + 8*t + 7*u, 5*y + 7*z + 8*t + 6*u, 16*z + 4*u, 16*z + 7*u, 15*u,
    7*u],
[14*u, 8*u, 13*x + 2*y + 4*t + 14*u, 4*y + 8*t + 2*u, 16*t + 6*u, 3*u],
[15*z + 9*t + 11*u, 2*y + 13*z + 8*t + 14*u, 7*t + 16*u, 8*x + 14*t + 6*
    11*t + 15*u, 11*z + 14*t + 16*u],
[12*z + 15*t + 6*u, 5*y + 7*z + 4*t + 9*u, t + u, 2*t + 2*u, 7*x + 4*t +
    12*z + t + 2*u],
[12*y + 10*z + 14*t + 14*u, 7*y + 15*z + 8*t, 4*u, 8*u, 16*u, 5*x + 15*z
    + 9*u],
[15*x, 15*x, 4*u, 8*u, 16*u, 0],
[14*z + 15*t + 9*u, 3*y + 11*z + 7*t + 13*u, 0, 11*z + 6*u, 2*z + 14*u,
    10*t + 4*u],
[10*z + 14*t + 14*u, 4*x + 12*y + 15*z + 8*t, 0, 0, 0, 15*z + 16*t + 9*u
[5*z + 16*t + 4*u, 12*y + 10*z + 15*t + 5*u, 0, 2*z + 15*u, 6*u, 15*z +
    7*u],
[13*z + t + 15*u, 4*y + 9*z + 16*t + 16*u, 0, 0, 0, 5*z + 11*t + 3*u]
```

```
> CohomologyDimension(K_X,0,0);
0
```

Finally, we verify some more cases of Serre duality which gives

$$\dim H^r(O_X(n)) = \dim H^{2-r}(K_X(-n)).$$

```
> [CohomologyDimension(K_X,0,i) eq CohomologyDimension(O_X,2,-i) :
>    i in [-1..5]];
[ true, true, true, true, true, true, true ]
```

109.16 Bibliography

[**AL94**] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases*, volume 3 of *Graduate studies in mathematics.* American Mathematical Society, Providence, R.I., 1994.

[**BS92**] David Bayer and Michael Stillman. Computation of Hilbert Functions. *J. Symbolic Comp.*, 14(1):31–50, 1992.

[**CLO98**] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.

[**DE02**] Wolfram Decker and David Eisenbud. Sheaf algorithms using the Exterior algebra. In Eisenbud et al., editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of *Springer Algorithms and Computation in Mathematics Series*, pages 215–247. Springer-Verlag, 2002.

[**DL06**] Wolfram Decker and Christoph Lossen. *Computing in Algebraic Geometry*, volume 16 of *Algorithms and Computation in Mathematics.* Springer, New York–Berlin–Heidelberg, 2006.

[**EFS03**] Eisenbud, Floystad, and Schreyer. Sheaf Cohomology and Free Resolutions over Exterior Algebras. *Trans. Am. Maths. Soc.*, 355:4397–4426, 2003.

[**Eis95**] David Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry*, volume 150 of *Graduate Texts in Mathematics.* Springer, New York–Berlin–Heidelberg, 1995.

[**Fau99**] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139 (1-3):61–88, 1999.

[**GP02**] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra.* Springer-Verlag, Berlin–Heidelberg–New York, 2002.

[**SS98**] Roberto La Scala and Michael Stillman. Strategies for Computing Minimal Free Resolutions. *J. Symbolic Comp.*, 26(4):409–431, 1998.

[**ST02**] Frank-Olaf Schreyer and Fabio Tonoli. Needles in a Haystack: Special Varieties via Small Fields. In Eisenbud et al., editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of *Springer Algorithms and Computation in Mathematics Series*, pages 251–277. Springer-Verlag, 2002.

110 INVARIANT THEORY

Chapter 110
INVARIANT THEORY

110.1 Introduction

Magma contains a powerful module for computing with invariant rings and fields of finite groups and algebraic groups. The algorithms for invariant theory of finite groups in Magma are based on those in the *Invar* package written in Maple, implemented by G. Kemper [Kem96], but also include many new ideas and improvements which are described in detail in a subsequent paper [KS97]. Since a detailed understanding of the latter paper is useful for better understanding of many of the functions in the chapter, it is recommended the paper be perused by anyone wishing to make serious applications of the functions.

Since V2.14, Magma also has algorithms for invariant theory of linear algebraic groups. In particular, Derksen's algorithm [Der99] and the algorithm by Beth and Müller-Quade [MQB99] have been implemented. These additions use code written by G. Kemper.

The primary goal of invariant theory in Magma is the computation of generators of the invariant ring or field of a given group, which may be finite or algebraic. The ground field may have arbitrary characteristic. In invariant theory of finite groups, the *modular case*, i.e., the case where the characteristic of the ground field K divides the group order, is of particular interest, since in that case there are still many theoretical questions unanswered. Magma also contains easy algorithms to calculate properties of modular invariant rings, such as the Hilbert series, the Cohen-Macaulay property, depth, and free resolutions.

The approach to calculating the invariant ring of a finite group is broken up into two major steps: first a system of *primary invariants* is constructed, i.e., homogeneous invariants $f_1, \ldots, f_n$ which are algebraically independent, such that the invariant ring is a finitely generated module over $A = K[f_1, \ldots, f_n]$. In the next step we calculate *secondary invariants*, which are generators of the invariant ring as an A-module.

Throughout this chapter, K will be a field and G is a group acting linearly on the n-dimensional vector space $V \cong K^n$ with basis $x_1, \ldots, x_n$. G may be a linear algebraic group, in which case K is assumed to be algebraically closed, or a finite matrix group, or a permutation group. G also acts on the symmetric algebra $K[V] = S(V)$, which is the multivariate polynomial ring $K[x_1, \ldots, x_n]$ in the variables $x_1, \ldots, x_n$. The invariant ring $R = \{f \in K[V] \mid f^\sigma = f \; \forall \sigma \in G\}$ is denoted by $K[V]^G$. The G-action extends naturally to the rational function field $K(V)$ on V, leading to the analogous definition of the invariants field $K(V)^G$.

Sections 110.2.1 through 110.7 describe the general setup of invariant theory in Magma. Section 110.8 is about computing invariants of specified degree. Sections 110.9 through 110.16 deal with functions for invariant rings of finite groups. The following Sections 110.17 and 110.18 present some functions whose scope is not limited to the context of invariant theory. Sections 110.20 and 110.21.3 are about functions for invariant rings of algebraic groups and for invariant fields, respectively. The Section 110.19 gives information about some low-level control of the data structures associated to invariant theory. Finally, since V2.14, the Section 110.20 deals with invariant rings of algebraic groups and the Section 110.21.3 deals with invariant fields.

110.2 Invariant Rings of Finite Groups

110.2.1 Creation

Let G be a finite matrix or permutation group acting on the polynomial ring $P = K[x_1, \ldots, x_n]$ over the field K. Magma allows the construction of the invariant ring $R = K[V]^G$. The invariant ring R is a special structure which contains references to the group G and polynomial ring P. When the invariant ring R is created using the `InvariantRing` function, no explicit calculations are done until specifically invoked (e.g., by the `PrimaryInvariants` function). The elements of R are the polynomials of P which are invariant under the action of G. Note that the parent of such polynomials is still P – the invariant ring R is just a special structure which contains all the information about the invariant ring. The category of invariant rings is `RngInvar`.

```
InvariantRing(G)
```

```
InvariantRing(G, K)
```

Construct the invariant ring $R = K[V]^G$ of the finite matrix or permutation group G over the field K. For a matrix group G, G alone should be supplied, while for a permutation group G, G should be supplied, together with the field K. The appropriate multivariate polynomial ring P is automatically constructed. No other explicit calculations are done (e.g. computation of primary invariants).

110.2.2 Access

The following functions allow simple access to basic properties of invariant rings.

```
Group(R)
```

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the group G.

```
CoefficientRing(R)
```

```
CoefficientField(R)
```

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the coefficient field K.

```
PolynomialRing(R)
```

Given an invariant ring $R = K[V]^G$ of the group G of degree n over the field K, return the polynomial ring $P = K[x_1, \ldots, x_n]$ in which the invariants of R lie. P has the print names `"x1", "x2"`, etc. – the angle bracket notation or the . operator should be used to assign the variables of P to actual Magma variables.

```
f in R
```

Return whether the polynomial f is in $R = K[V]^G$. Note that the parent of f is always the polynomial ring P, never R, so a `true` result does not mean that the parent of f is R.

110.3 Group Actions on Polynomials

This section describes in detail the actions which groups have on multivariate polynomial rings.

110.4 Permutation Group Actions on Polynomials

If P is a polynomial ring in n indeterminates $x_1, \ldots, x_n$, over any coefficient ring, $\mathrm{Sym}(n)$ acts on P by permuting the indices of the indeterminates. Thus, the polynomial $f(x_1, \ldots, x_n)$ is mapped into the polynomial $f(x_{g(1)}, \ldots, x_{g(n)})$.

```
f ^ g
```

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation g belonging to a subgroup of $\mathrm{Sym}(\{1, \ldots, n\})$, return the image of f under g.

```
f ^ G
```

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation group G contained in $\mathrm{Sym}(\{1, \ldots, n\})$, return the orbit of f under G.

```
IsInvariant(f, g)
```

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation g of degree n or an element of a matrix group of degree n whose coefficient ring is the same as that of f, return whether f is an invariant of g, i.e., whether $f^g = f$.

```
IsInvariant(f, G)
```

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation group G of degree n or a matrix group of degree n whose coefficient ring is the same as that of f, return whether f is an invariant of G, i.e., whether $f^g = f$ for all $g \in G$.

110.5 Matrix Group Actions on Polynomials

If P is a polynomial ring in n indeterminates $x_1, \ldots, x_n$, over the ring S, then $\mathrm{GL}(n, S)$ acts on P as follows: Let $\mathbf{x}$ denote the vector $(x_1, \ldots, x_n)$. Then the image g of a polynomial f of P under the action of a matrix a of $\mathrm{GL}(n, S)$ is defined by $g(\mathbf{x}) = f(\mathbf{x} * a)$.

`f ^ a`

Given a polynomial f belonging to a polynomial ring having n indeterminates and coefficient ring S, and a matrix a belonging subgroup G of $\mathrm{GL}(n, S)$, return the image of f under a.

`f ^ G`

Given a polynomial f belonging to a polynomial ring having n indeterminates and coefficient ring S, and a to a subgroup of $\mathrm{GL}(n, S)$, return the orbit of f under G.

Example H110E1

We act on the polynomial ring in two indeterminates over the field $K = Q(\sqrt{2})$, by a cyclic subgroup of $\mathrm{GL}(2, K)$.

```
> K := QuadraticField(2);
> Aq := [ x / K.1 : x in [1, 1, -1, 1]];
> G := MatrixGroup<2, K | Aq>;
> P<x, y> := PolynomialRing(K, 2);
> f := x^2 + x * y + y^2;
> g := f^G.1;
> g;
1/2*x^2 + 3/2*y^2
> f^G;
{
   1/2*x^2 + 3/2*y^2,
   x^2 - x*y + y^2,
   x^2 + x*y + y^2,
   3/2*x^2 + 1/2*y^2
}
```

110.6 Algebraic Group Actions on Polynomials

In the invariant theory package of Magma, a linear algebraic group G is given by polynomials, say in variables $t_1, \ldots, t_m$, defined over some field K that is representable in Magma, as the affine variety over the algebraic closure $\bar{K}$ of K given by these polynomials. A G-module is given by a matrix $A \in K[t_1, \ldots, t_m]^{n \times n}$ such that a group element $(\eta_1, \ldots, \eta_m) \in G$ acts on $\bar{K}^n$ by the matrix obtained by substituting $(\eta_1, \ldots, \eta_m)$ into the polynomials occurring in the matrix A.

G then also acts on the ring of polynomials on $\bar{K}^n$ by

$$\sigma(f) = f \circ \sigma^{-1}$$

for $\sigma \in G$ and $f \in \bar{K}[x_1, \ldots, x_n]$. Since the algorithms in Magma do not work with the algebraic closure, single group elements are never dealt with. In fact, all relevant algorithms ony involve field elements of K, the field of definition.

110.7 Verbosity

The following procedure allows verbose information for the Invariant Theory algorithms to be displayed.

```
SetVerbose("Invariants", v)
```

(Procedure.) Set the verbose printing level for the Invariant Theory algorithms of Magma to be v. Currently the legal values for v are `true`, `false`, 0, 1, 2, 3, or 4 (`false` has the same effect as 0, and `true` has the same effect as 1). Level 1 gives a minimal amount of useful information during the running of all the algorithms while higher levels give more detailed information. For the primary invariants computation, the verbose output displays each possible degree list (degrees of the potential primary invariants) before and then tries to find primary invariants corresponding to this degree list. For the secondary invariants computation, in the non-modular case the algorithm loops over the necessary degrees in increasing order and computes the relevant new invariants; in the modular case the algorithm finds secondary invariants with respect to a subgroup and then performs a module syzygy computation. Full details of the algorithms are found in [KS97].

110.8 Construction of Invariants of Specified Degree

Let $R = K[V]^G$ be the invariant ring of the group G over the field K. Let $d \geq 0$ be a fixed integer. The homogeneous invariants in R of degree d form a vector space R_d over K.

There are two ways of explicitly constructing homogeneous invariants in R of degree d: the *Reynolds operator* method and the *linear algebra* method. Both methods are described in detail in [KS97].

The Reynolds operator method only works fro finite groups in the non-modular case. It takes a monomial of degree d and yields either the zero polynomial or a non-zero invariant of degree d. By applying it to several different monomials, a complete basis of R_d can be constructed. If G is a permutation group, a simplified version of the Reynolds operator can always be used which is independent of the field K (and thus whether we are in the modular case or not).

The linear algebra method works in both the modular and non-modular cases and, with appropriate modifications, also for linear algebraic groups. It simply finds a basis for R_d in one step – it is not possible to find a single invariant alone by this method.

Magma provides the function `InvariantsOfDegree` to automatically compute a basis of R_d by a default appropriate method – the method can also be selected by a parameter. The function `InvariantsOfDegree` can also be given a positive integer k which is less than or equal to the dimension of R_d: in such a case, only k linearly independent invariants are computed. See also the functions `MonomialsOfDegree` and `MonomialsOfWeightedDegree` in the Ideal Theory chapter.

```
ReynoldsOperator(f, G)
```

Given a polynomial f and a matrix group G such that G can act on f, return the application of the Reynolds operator of G to f. (f need not be a monomial but may be a non-homogeneous polynomial.)

```
InvariantsOfDegree(R, d)
InvariantsOfDegree(G, d)
InvariantsOfDegree(G, K, d)
InvariantsOfDegree(G, P, d)
```

`Invariants` MonStgElt *Default : "Both"*

Construct a K-basis of the space R_d of the homogeneous invariants of degree d in the invariant ring $R = K[V]^G$ of the group G over the field K as a sequence of polynomials. Either the invariant ring R, the group G (if a matrix group), or the group G (if a permutation group) together with the field K may be passed. A specific polynomial ring P compatible with G and K may be passed so that the returned invariants lie in P. The parameter `Invariants` may be supplied to select the method of the construction of the invariants: `"Reynolds"` (use the Reynolds operator), `"Linear"` (use the linear algebra method), or `"Both"` (use an appropriate combination of both methods). The default is `"Both"`.

```
InvariantsOfDegree(R, d, k)
InvariantsOfDegree(G, d, k)
InvariantsOfDegree(G, K, d, k)
InvariantsOfDegree(G, P, d, k)
```

`Invariants` MonStgElt *Default : "Both"*

Construct k linearly independent homogeneous invariants of degree d in the invariant ring $R = K[V]^G$ of the group G over the field K as a sequence of polynomials, where k must be greater than or equal to 1 and less than or equal to the dimension of the space R_d. Either the invariant ring R, the group G (if a matrix group), or the group G (if a permutation group) together with the field K may be passed. A specific polynomial ring P compatible with G and K may be passed so that the returned invariants lie in P. The parameter `Invariants` may be supplied to select the method of the construction of the invariants – see the last function.

Example H110E2

We demonstrate elementary uses of `ReynoldsOperator` and `InvariantsOfDegree`.

```
> K<z> := CyclotomicField(5);
> w := -z^3 - z^2;
> G := MatrixGroup<3,K |
>     [1,0,-w, 0,0,-1, 0,1,-w],
>     [-1,-1,w, -w,0,w, -w,0,1]>;
> P<x1,x2,x3> := PolynomialRing(K, 3);
> time ReynoldsOperator(x1^4, G);
(-z^3 - z^2 + 1)*x1^4 + (12/5*z^3 + 12/5*z^2 -
    4/5)*x1^3*x2 + (12/5*z^3 + 12/5*z^2 - 4/5)*x1^3*x3
    + (-14/5*z^3 - 14/5*z^2 + 14/5)*x1^2*x2^2 +
    (4/5*z^3 + 4/5*z^2 + 4/5)*x1^2*x2*x3 + (-14/5*z^3 -
    14/5*z^2 + 14/5)*x1^2*x3^2 + (12/5*z^3 + 12/5*z^2 -
    4/5)*x1*x2^3 + (4/5*z^3 + 4/5*z^2 + 4/5)*x1*x2^2*x3
    + (4/5*z^3 + 4/5*z^2 + 4/5)*x1*x2*x3^2 + (12/5*z^3
    + 12/5*z^2 - 4/5)*x1*x3^3 + (-z^3 - z^2 + 1)*x2^4 +
    (12/5*z^3 + 12/5*z^2 - 4/5)*x2^3*x3 + (-14/5*z^3 -
    14/5*z^2 + 14/5)*x2^2*x3^2 + (12/5*z^3 + 12/5*z^2 -
    4/5)*x2*x3^3 + (-z^3 - z^2 + 1)*x3^4
Time: 0.090
> time I20_1 := InvariantsOfDegree(G, 20, 1);
0.259
> time I20 := InvariantsOfDegree(G, 20);
3.589
> [LeadingMonomial(f): f in I20];
[
    x1^20,
    x1^18*x2^2,
    x1^16*x2^4,
    x1^15*x2^5,
    x1^14*x2^6,
    x1^13*x2^7,
    x1^12*x2^8
]
> G := CyclicGroup(4);
> K := GF(2);
> InvariantsOfDegree(G, K, 4);
[
    x1^4 + x2^4 + x3^4 + x4^4,
    x1^3*x2 + x1*x4^3 + x2^3*x3 + x3^3*x4,
    x1^3*x3 + x1*x3^3 + x2^3*x4 + x2*x4^3,
    x1^3*x4 + x1*x2^3 + x2*x3^3 + x3*x4^3,
    x1^2*x2^2 + x1^2*x4^2 + x2^2*x3^2 + x3^2*x4^2,
    x1^2*x2*x3 + x1*x2*x4^2 + x1*x3^2*x4 + x2^2*x3*x4,
    x1^2*x2*x4 + x1*x2^2*x3 + x1*x3*x4^2 + x2*x3^2*x4,
    x1^2*x3^2 + x2^2*x4^2,
    x1^2*x3*x4 + x1*x2^2*x4 + x1*x2*x3^2 + x2*x3*x4^2,
    x1*x2*x3*x4
]
```

`SetAllInvariantsOfDegree(R, d, Q)`

(Procedure.) Given an invariant ring $R = K[V]^G$, an integer $d \geq 0$, and a sequence Q consisting of k degree-d homogeneous invariants of G, set the internal list of all linearly-independent homogeneous invariants of degree d of R to be Q. Thus the elements of Q must describe a basis of the space of all homogeneous invariants of degree d of R. If the Hilbert Series of R is known, it will be used to check that the length of Q (the dimension of the basis) is correct.

Example H110E3

We demonstrate a simple use of SetAllInvariantsOfDegree.

```
> R := InvariantRing(CyclicGroup(4), GF(2));
> P<x1,x2,x3,x4> := PolynomialRing(R);
> L := [
>     x1^2 + x2^2 + x3^2 + x4^2,
>     x1*x2 + x1*x4 + x2*x3 + x3*x4,
>     x1*x3 + x2*x4
> ];
> SetAllInvariantsOfDegree(R, 2, L);
> InvariantsOfDegree(R, 2);
[
    x1^2 + x2^2 + x3^2 + x4^2,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4
]
> PrimaryInvariants(R);
[
    x1 + x2 + x3 + x4,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4,
    x1*x2*x3*x4
]
```

The following sections 110.9 through 110.16 all deal with invariant rings of finite groups.

110.9 Construction of G-modules

This section describes how one can create a finite-dimensional G-module corresponding to the action of a finite group G on a polynomial ring P. There are two ways one can create a finite-dimensional action: the action on the space of homogeneous polynomials of a fixed degree, or the action on the quotient space of polynomials by a zero-dimensional ideal (so the quotient has finite-dimension as a vector space). The functions in this section are also found in the chapter on general modules but are also included here since they are useful in Invariant Theory.

`GModule(G, P, d)`

Given a finite permutation or matrix group G of degree n, a polynomial ring $P = K[x_1, \ldots, x_n]$ over a field K, and a non-negative integer d, create the $K[G]$-module M corresponding to the action of G on the space of homogeneous polynomials of degree d of the polynomial ring P. The function also returns the isomorphism f between the space of homogeneous polynomials of degree d of P and M, together with an indexed set of monomials of degree d of P which correspond to the columns of M.

`GModule(G, I, J)`

Given a finite permutation or matrix group G of degree n, an ideal I of a multivariate polynomial ring $P = K[x_1, \ldots, x_n]$ over a field K, and a zero-dimensional subideal J of I, create the $K[G]$-module M corresponding to the action of G on the finite-dimensional quotient I/J. The function also returns the isomorphism f between the quotient space I/J and M, together with an indexed set of monomials of P, forming a (vector space) basis of I/J, and which correspond to the columns of M.

`GModule(G, Q)`

Given a finite permutation or matrix group G of degree n, and a finite-dimensional quotient ring $Q = I/J$ of a multivariate polynomial ring $P = K[x_1, \ldots, x_n]$ over a field K, create the $K[G]$-module M corresponding to the action of G on the finite-dimensional quotient Q. The function also returns the isomorphism f between the quotient ring Q and M, together with an indexed set of monomials of P, forming a (vector space) basis of Q, and which correspond to the columns of M.

Example H110E4

We demonstrate simple uses of the `GModule` function.

```
> q := 5;
> K := GF(q);
> G := GL(3, K);
> P<x, y, z> := PolynomialRing(K, 3);
> I := ideal< P | x^5 - x,y^5 - y, z^5 - z >;
> Q, rho := quo< P | I >;
> f := x^3 + x^2*y + y^3;
> M, phi:= GModule(G, P, I);
> Constituents(M);
[
    GModule of dimension 1 over GF(5),
    GModule of dimension 3 over GF(5),
    GModule of dimension 3 over GF(5),
    GModule of dimension 6 over GF(5),
    GModule of dimension 6 over GF(5),
    GModule of dimension 10 over GF(5),
    GModule of dimension 10 over GF(5),
    GModule of dimension 15 over GF(5),
    GModule of dimension 15 over GF(5),
    GModule of dimension 18 over GF(5),
    GModule of dimension 18 over GF(5),
    GModule of dimension 19 over GF(5)
]
> N := sub<M | phi(f)>;
> N;
GModule N of dimension 10 over GF(5)
> M5 := GModule(G, P, 5);
> M5;
GModule M5 of dimension 21 over GF(5)
> Constituents(M5);
[
    GModule of dimension 3 over GF(5),
    GModule of dimension 18 over GF(5)
]
```

110.10 Molien Series

Let $R = K[V]^G$ be the invariant ring of the finite group G over the field K. If G is a finite matrix group in the non-modular case or a permutation group (in *either* the modular or non-modular case) then the Molien series of G yields the Hilbert Series of R.

`MolienSeries(G)`

The Molien series of G, returned as an element of the rational function field $\mathbf{Z}(t)$. If G is a permutation group, the Molien series always exists and equals the Hilbert series of the invariant ring of G for any field. If G is a matrix group, the characteristic of the coefficient field of G must be coprime with the order of G.

`MolienSeriesApproximation(G, n)`

The Molien series of a permutation group G, or more precisely, an approximation to it, as a Laurent series with n known coefficients. In contrast to the `MolienSeries` function above, approximations can be computed for far larger groups.

Example H110E5

We compute the Molien series of a matrix G and verify that the coefficients of the corresponding power series match the number of independent invariants for each degree.

```
> K<z> := CyclotomicField(5);
> w := -z^3 - z^2;
> G := MatrixGroup<3,K |
>     [1,0,-w, 0,0,-1, 0,1,-w],
>     [-1,-1,w, -w,0,w, -w,0,1]>;
> M<t> := MolienSeries(G);
> M;
(-t^8 - t^7 + t^5 + t^4 + t^3 - t - 1)/(t^11 + t^10 -
    t^9 - 2*t^8 - t^7 + t^4 + 2*t^3 + t^2 - t - 1)
> P<u> := PowerSeriesRing(IntegerRing());
> P ! M;
1 + u^2 + u^4 + 2*u^6 + 2*u^8 + 3*u^10 + 4*u^12 +
    4*u^14 + u^15 + 5*u^16 + u^17 + 6*u^18 + u^19 +
    O(u^20)
> Coefficients(P ! M);
[ 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 4, 0, 4, 1, 5, 1,
6, 1 ]
> time [#InvariantsOfDegree(G, i): i in [0 .. 19]];
[ 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 4, 0, 4, 1, 5, 1,
6, 1 ]
```

110.11 Primary Invariants

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. A set of *primary invariants* of R is a set $\{f_1, \ldots, f_n\}$ of n algebraically independent homogeneous invariants of R such that the invariant ring R is a finitely generated module over $A = K[f_1, \ldots, f_n]$. A set of primary invariants always exists for any invariant ring R. The invocation `PrimaryInvariants(R)` allows automatic construction of primary invariants of R. The primary invariants are stored in R and recalled as necessary in subsequent computations.

The latest algorithm in Magma to compute primary invariants, due to G. Kemper [Kem99], now guarantees that the degrees of the primary invariants found by the algorithm are optimal (with respect to their product and then their sum).

```
PrimaryInvariants(R)
```

Construct optimal primary invariants for the invariant ring $R = K[V]^G$ as a sorted sequence (with increasing degrees) of n polynomials of R where n is the degree of G.

Example H110E6

We compute primary invariants for the "first A_5 in $\mathrm{SL}(\mathbf{F}_2)$", discussed in [AM94, p. 116]. The resulting degrees $3, 5, 8$, and 12 are necessarily optimal (see [Kem96]).

```
> K := GF(2);
> G := MatrixGroup<4, K |
>     [0,1,0,0, 1,1,0,0, 0,0,1,1, 0,0,1,0],
>     [1,0,0,0, 0,1,0,0, 1,0,1,0, 0,1,0,1],
>     [1,0,1,0, 0,1,0,1, 0,0,1,0, 0,0,0,1]>;
> R := InvariantRing(G);
> time p := PrimaryInvariants(R);
Time: 1.399
> [TotalDegree(f): f in p];
[ 3, 5, 8, 12 ]
```

110.12 Secondary Invariants

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. If $\{f_1, \ldots, f_n\}$ is a set of primary invariants for R then R can be viewed as a finitely generated module over the algebra $A = K[f_1, \ldots, f_n]$. A set of *secondary invariants* for R with respect to these primary invariants is set of module generators over A. The invocation `SecondaryInvariants(R)` allows automatic construction of secondary invariants of R. The secondary invariants are stored in R and recalled as necessary in subsequent computations. Different algorithms are needed for the modular and non-modular cases – see [KS97] for details.

```
SecondaryInvariants(R)
```

Construct secondary invariants for the invariant ring $R = K[V]^G$ (with respect to the current primary invariants of R, constructed automatically first if necessary) as a sorted sequence (with increasing degrees) of polynomials of R. The secondary invariants are *minimal*; i.e. they are a minimal generating set for R considered as a module over the algebra generated by the primary invariants.

```
SecondaryInvariants(R, H)
```

Construct secondary invariants for the *modular* invariant ring $R = K[V]^G$ (with respect to the current primary invariants of R), using the subgroup H. This function can only be used if R is a modular invariant ring. H must be a subgroup of the group G; first, secondary invariants are computed for $K[V]^H$ using the current primary invariants for G and then these secondary invariants are used in the manner described in [KS97]. The function `SecondaryInvariants(R)` (taking just the invariant ring R) follows a default strategy in which it tries to use this function with the best subgroup H appropriate. Thus usually using this function to specify a particular subgroup is not more helpful than the one-argument function but occasionally it may be.

```
IrreducibleSecondaryInvariants(R)
```

Return the irreducible secondary invariants of the invariant ring $R = K[V]^G$ (with respect to the current primary invariants of R, constructed automatically first if necessary) as a sequence of polynomials of R. These, together with the primary invariants of R, generate R as an algebra over K. In the modular case, these will be the same as the secondary invariants of R (excluding the polynomial 1) but in the non-modular case they may form a proper subsequence of the secondary invariants. Note that the expression of the secondary invariants in terms of the irreducible secondary invariants is given as the second return value of the function `Algebra` (see the section on the algebra of an invariant ring and algebraic relations below).

Example H110E7

We construct primary and then secondary invariants for the invariant ring R of the group G over $\mathbf{F}_2$, where G is the (permutation) cyclic group of order 4. Note that in this example Noether's degree bound (which holds for characteristic 0) is violated.

```
> K := GF(2);
> G := CyclicGroup(4);
> R := InvariantRing(G, K);
> time PrimaryInvariants(R);
[
    x1 + x2 + x3 + x4,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4,
    x1*x2*x3*x4
]
Time: 0.040
> time SecondaryInvariants(R);
[
    1,
    x1*x2*x3 + x1*x2*x4 + x1*x3*x4 + x2*x3*x4,
    x1^2*x3 + x1^2*x4 + x1*x2^2 + x1*x3^2 + x2^2*x4 +
        x2*x3^2 + x2*x4^2 + x3*x4^2,
    x1^2*x3^2 + x1^2*x3*x4 + x1*x2^2*x4 + x1*x2*x3^2 +
        x2^2*x4^2 + x2*x3*x4^2,
    x1^3*x3*x4 + x1^2*x2^2*x3 + x1^2*x2^2*x4 +
        x1^2*x2*x3^2 + x1^2*x2*x3*x4 + x1^2*x2*x4^2 +
        x1^2*x3^2*x4 + x1^2*x3*x4^2 + x1*x2^3*x4 +
        x1*x2^2*x3^2 + x1*x2^2*x3*x4 + x1*x2^2*x4^2 +
        x1*x2*x3^3 + x1*x2*x3^2*x4 + x1*x2*x3*x4^2 +
        x1*x3^2*x4^2 + x2^2*x3^2*x4 + x2^2*x3*x4^2 +
        x2*x3^2*x4^2 + x2*x3*x4^3
]
Time: 0.080
```

110.13 Fundamental Invariants

Let $R = K[V]^G$ be the invariant ring of the group G over the field K and suppose the degree of G is n. A set of *fundamental invariants* for R is a generating set of R as an algebra over K.

```
FundamentalInvariants(R)
```

`Al`	MonStgElt	*Default : "King"*
`MaxDegree`	RngIntElt	*Default :* 0

Construct fundamental invariants for the invariant ring $R = K[V]^G$ as a sorted sequence (with increasing degrees) of polynomials of R.

As of V2.15, if R is non-modular, then by default the fundamental invariants are computed via the algorithm of S.King [Kin07]; the alternative algorithm (always used in the modular case), which computes the fundamental invariants by minimalizing the union of the primary and secondary invariants of R, may be selected by setting the parameter `Al` to `"MinPrimSec"`.

If the fundamental invariants are known to be bounded by degree d, then the parameter `MaxDegree` may be set to d to assist the King algorithm with an early stopping condition in the non-modular case.

Example H110E8

We construct fundamental invariants for the invariant ring R of the group G over **Q**, where G is permutation group consisting of two parallel copies of S_3 in degree 6. Notice that the sequence of fundamental invariants is shorter and simpler than the sequence consisting of the primary invariants combined with the secondary invariants.

```
> K := RationalField();
> G := PermutationGroup<6 | (1,2,3)(4,5,6), (1,2)(4,5)>;
> R := InvariantRing(G, K);
> PrimaryInvariants(R);
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x4^3 + x5^3 + x6^3
]
> SecondaryInvariants(R);
[
    1,
    x1*x4 + x2*x5 + x3*x6,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x1^2*x4^2 + 2*x1*x2*x4*x5 + 2*x1*x3*x4*x6 + x2^2*x5^2 + 2*x2*x3*x5*x6 +
        x3^2*x6^2,
    x1^3*x4^3 + x1^2*x2*x4*x5^2 + x1^2*x3*x4*x6^2 + x1*x2^2*x4^2*x5 +
        x1*x3^2*x4^2*x6 + x2^3*x5^3 + x2^2*x3*x5*x6^2 + x2*x3^2*x5^2*x6 +
        x3^3*x6^3
]
> FundamentalInvariants(R);
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x1*x4 + x2*x5 + x3*x6,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x4^3 + x5^3 + x6^3
]
```

Example H110E9

As in [Kin07], we compute fundamental invariants for the invariant rings for all transitive groups of degree 7 (in characteristic zero). For each group, we print its order and a summary of the degrees (where the i-th element of the sequence gives the number of fundamental invariants of degree i).

```
> function deg_summary(B)
>     degs := [TotalDegree(f): f in B];
>     return [#[j: j in degs | j eq d]: d in [1 .. Max(degs)]];
> end function;
>
> d := 7;
> time for i := 1 to NumberOfTransitiveGroups(d) do
>     G := TransitiveGroup(d, i);
>     R := InvariantRing(G, RationalField());
>     F := FundamentalInvariants(R);
>     printf "%o: Order: %o, Degrees: %o\n", i, #G, deg_summary(F);
> end for;
1: Order: 7, Degrees: [ 1, 3, 8, 12, 12, 6, 6 ]
2: Order: 14, Degrees: [ 1, 3, 4, 6, 6, 3, 3 ]
3: Order: 21, Degrees: [ 1, 1, 4, 5, 8, 8, 6 ]
4: Order: 42, Degrees: [ 1, 1, 2, 3, 4, 7, 7, 5, 1 ]
5: Order: 168, Degrees: [ 1, 1, 2, 2, 2, 2, 2 ]
6: Order: 2520, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
7: Order: 5040, Degrees: [ 1, 1, 1, 1, 1, 1, 1 ]
Time: 1.610
```

Instead of computing over the rational field, for each group G we can instead compute over $\mathbf{F}_p$, where p is the smallest prime which does not divide the order of G. This is faster, and it is conjectured that the resulting degrees are always the same as for the computation over the rationals.

```
> d := 7;
> time for i := 1 to NumberOfTransitiveGroups(d) do
>     G := TransitiveGroup(d, i);
>     p := rep{p: p in [2 .. #G] | IsPrime(p) and #G mod p ne 0};
>     R := InvariantRing(G, GF(p));
>     F := FundamentalInvariants(R);
>     printf "%o: Order: %o, Degrees: %o\n", i, #G, deg_summary(F);
> end for;
1: Order: 7, Degrees: [ 1, 3, 8, 12, 12, 6, 6 ]
2: Order: 14, Degrees: [ 1, 3, 4, 6, 6, 3, 3 ]
3: Order: 21, Degrees: [ 1, 1, 4, 5, 8, 8, 6 ]
4: Order: 42, Degrees: [ 1, 1, 2, 3, 4, 7, 7, 5, 1 ]
5: Order: 168, Degrees: [ 1, 1, 2, 2, 2, 2, 2 ]
6: Order: 2520, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
7: Order: 5040, Degrees: [ 1, 1, 1, 1, 1, 1, 1 ]
Time: 0.790
```

Finally, we can do the same for all transitive groups of degree 8 in about 2 minutes.

```
> d := 8;
> time for i := 1 to NumberOfTransitiveGroups(d) do
>     G := TransitiveGroup(d, i);
>     p := rep{p: p in [2 .. #G] | IsPrime(p) and #G mod p ne 0};
>     R := InvariantRing(G, GF(p));
>     F := FundamentalInvariants(R);
>     printf "%o: Order: %o, Degrees: %o\n", i, #G, deg_summary(F);
> end for;
1: Order: 8, Degrees: [ 1, 4, 10, 18, 16, 8, 4, 4 ]
2: Order: 8, Degrees: [ 1, 5, 9, 16, 8 ]
3: Order: 8, Degrees: [ 1, 7, 7, 7 ]
4: Order: 8, Degrees: [ 1, 6, 8, 12, 5 ]
5: Order: 8, Degrees: [ 1, 4, 10, 19, 15, 7 ]
6: Order: 16, Degrees: [ 1, 4, 5, 9, 8, 4, 2, 2 ]
7: Order: 16, Degrees: [ 1, 3, 7, 12, 13, 9, 4, 4 ]
8: Order: 16, Degrees: [ 1, 3, 6, 11, 12, 7, 2, 2 ]
9: Order: 16, Degrees: [ 1, 5, 5, 8, 4 ]
10: Order: 16, Degrees: [ 1, 4, 6, 11, 7, 2 ]
11: Order: 16, Degrees: [ 1, 4, 6, 11, 7, 3 ]
12: Order: 24, Degrees: [ 1, 2, 4, 8, 11, 12, 7 ]
13: Order: 24, Degrees: [ 1, 3, 3, 7, 8, 11, 7 ]
14: Order: 24, Degrees: [ 1, 3, 3, 8, 7, 9, 6, 1, 1 ]
15: Order: 32, Degrees: [ 1, 3, 4, 7, 6, 4, 2, 2 ]
16: Order: 32, Degrees: [ 1, 3, 5, 8, 7, 7, 4, 4 ]
17: Order: 32, Degrees: [ 1, 3, 4, 7, 6, 4, 2, 2 ]
18: Order: 32, Degrees: [ 1, 4, 4, 7, 3 ]
```

```
19: Order: 32, Degrees: [ 1, 3, 3, 7, 6, 7, 5, 1 ]
20: Order: 32, Degrees: [ 1, 3, 5, 9, 6, 4, 2, 1 ]
21: Order: 32, Degrees: [ 1, 4, 4, 6, 4, 3, 2, 1 ]
22: Order: 32, Degrees: [ 1, 4, 4, 7, 3, 1 ]
23: Order: 48, Degrees: [ 1, 2, 3, 5, 6, 6, 5, 2 ]
24: Order: 48, Degrees: [ 1, 3, 3, 6, 4, 3, 1 ]
25: Order: 56, Degrees: [ 1, 1, 1, 4, 6, 13, 18, 23, 18, 6 ]
26: Order: 64, Degrees: [ 1, 3, 3, 5, 3, 3, 2, 3, 1 ]
27: Order: 64, Degrees: [ 1, 3, 5, 8, 6, 4, 2, 2 ]
28: Order: 64, Degrees: [ 1, 3, 3, 5, 4, 4, 2, 2 ]
29: Order: 64, Degrees: [ 1, 3, 3, 6, 3, 2, 1 ]
30: Order: 64, Degrees: [ 1, 3, 3, 5, 3, 2, 3, 4, 3, 2, 1, 1 ]
31: Order: 64, Degrees: [ 1, 4, 4, 6, 3, 1 ]
32: Order: 96, Degrees: [ 1, 2, 2, 4, 3, 5, 4, 2, 2, 1, 1, 1 ]
33: Order: 96, Degrees: [ 1, 2, 2, 4, 3, 6, 5, 5, 3 ]
34: Order: 96, Degrees: [ 1, 2, 2, 5, 2, 5, 4, 3, 3 ]
35: Order: 128, Degrees: [ 1, 3, 3, 5, 3, 2, 1, 1 ]
36: Order: 168, Degrees: [ 1, 1, 1, 2, 2, 5, 6, 8, 10, 11, 8 ]
37: Order: 168, Degrees: [ 1, 1, 1, 3, 1, 5, 5, 8, 9, 9, 7 ]
38: Order: 192, Degrees: [ 1, 2, 2, 3, 3, 5, 4, 3, 2, 1, 1, 1 ]
39: Order: 192, Degrees: [ 1, 2, 2, 4, 2, 2, 1 ]
40: Order: 192, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1, 1, 3, 3, 2, 2, 1, 1, 1 ]
41: Order: 192, Degrees: [ 1, 2, 2, 4, 2, 3, 2, 2, 1 ]
42: Order: 288, Degrees: [ 1, 2, 2, 3, 2, 3, 2, 2, 1, 1 ]
43: Order: 336, Degrees: [ 1, 1, 1, 2, 1, 3, 3, 5, 4, 6, 5, 4, 2 ]
44: Order: 384, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1 ]
45: Order: 576, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1, 0, 0, 0, 1 ]
46: Order: 576, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1 ]
47: Order: 1152, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1 ]
48: Order: 1344, Degrees: [ 1, 1, 1, 2, 1, 2, 2, 2, 1, 1 ]
49: Order: 20160, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
50: Order: 40320, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 1 ]
Time: 128.030
```

Example H110E10

We compute fundamental invariants of a degree-10 representation of S_5 acting on pairs. See [Kin07, p.11–12]. First we compute the fundamental invariants mod 7 of the permutation representation (very difficult in practice hitherto).

```
> G := PermutationGroup<10 | (2,5)(3,6)(4,7),(1,5,8,10,4)(2,6,9,3,7)>;
> #G;
120
> R := InvariantRing(G, GF(7));
> time F := FundamentalInvariants(R);
Time: 29.310
> {* Degree(f): f in F *};
```

```
{* 1, 2^^2, 3^^4, 4^^7, 5^^10, 6^^13, 7^^13, 8^^4, 9^^2 *}
```

Finally, we can compute a matrix representation of G as a direct sum of irreducible representations of degrees 1, 4 and 5. We then compute the fundamental invariants of the invariant ring of this representation mod 7.

```
> Q := RationalField();
> R0 := InvariantRing(G, Q);
> P0 := PolynomialRing(R0);
> M := GModule(G, Q);
> Gl := MatrixGroup(M);
> C := CharacterTable(Gl);
> Pi := [&+[Q!Integers()!c(g)*MatrixAlgebra(Q, 10)!g: g in Gl]/#G: c in C];
> Pi := [p: p in Pi | p ne 0];
> L := [sub<M | Image(p)>: p in Pi];
> G := MatrixGroup(DirectSum(DirectSum(L[1],L[2]),L[3]));
> G;
MatrixGroup(10, Rational Field)
Generators:
    [   1    0    0    0    0    0    0    0    0    0]
    [   0    1  1/3  1/3  1/3    0    0    0    0    0]
    [   0    0  1/3 -2/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3  1/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3 -2/3  1/3    0    0    0    0    0]
    [   0    0    0    0    0    1    0    0    0    0]
    [   0    0    0    0    0    0    0    0    1    0]
    [   0    0    0    0    0    0    0    0    0    1]
    [   0    0    0    0    0    0    1    0    0    0]
    [   0    0    0    0    0    0    0    1    0    0]
    [   1    0    0    0    0    0    0    0    0    0]
    [   0    0  1/3 -2/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3  1/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3 -2/3  1/3    0    0    0    0    0]
    [   0    1  1/3  1/3  1/3    0    0    0    0    0]
    [   0    0    0    0    0   -1   -1    1    1    0]
    [   0    0    0    0    0   -1    0    0    0    1]
    [   0    0    0    0    0   -1    0    1    0    0]
    [   0    0    0    0    0    0   -1    0    0    0]
    [   0    0    0    0    0    0   -1    1    0    0]
> Gp := ChangeRing(G, GF(7));
> #Gp;
120
> Rp := InvariantRing(Gp);
> time Fp := FundamentalInvariants(Rp);
Time: 35.380
> {* Degree(f): f in Fp *};
{* 1, 2^^2, 3^^4, 4^^7, 5^^10, 6^^13, 7^^13, 8^^4, 9^^2 *}
> [Degree(f): f in F] eq [Degree(f): f in Fp];
```

```
true
```

110.14 The Module of an Invariant Ring

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. Suppose also that primary invariants $\{f_1, \ldots, f_n\}$ for R have been constructed, together with minimal secondary invariants $S = \{g_1, \ldots, g_m\}$ for R with respect to these primary invariants. (These secondary invariants may possess non-trivial module syzygies.) Then R can be considered as a module over the algebra $A = K[f_1, \ldots, f_n]$ with the minimal (module) generating set S.

To compute with this module structure of R easily, Magma automatically constructs the graded multivariate polynomial algebra $A' = K[t_1, \ldots, t_n]$ (with the weighted degree of the variable t_i defined to be the degree of f_i) which is isomorphic to A, and then constructs the graded module $M = A'^m/Q$ over A' with the quotient relations Q given by the syzygies of the g_i (and with the weighted degree of column i equal to the degree of g_i). The algebra A' is isomorphic to A under the map $t_i \mapsto f_i$, and the module M is isomorphic to R (considered as a module) under the map $M.i \mapsto g_i$ (extended by the isomorphism from A' onto A). (See the chapter on modules over $K[x_1, \ldots, x_n]$ for details on how to compute with the module M and an explanation of quotient relations, the unit vectors $M.i$, etc.) Once the module M is created, together with the isomorphism $f : R \to M$, one can apply f to a general element h of R to obtain the element of M corresponding to h. This effectively yields a representation of h as a sum $\sum_{i=1} ka_i g_i$ with $a_i \in A$ in terms of the primary and secondary invariants. This representation is also unique up to the relations given by the syzygies of the g_i.

When creating the module M, the coefficient ring A' of M is assigned the print names "t1", "t2", etc. – the angle bracket notation or the . operator should be used to assign the variables of A' to actual Magma variables.

```
Module(R)
```

The module M isomorphic to $R = K[V]^G$, together with the isomorphism $f : R \to M$.

Example H110E11

We create the module M corresponding to the invariant ring R of the group G generated by the 4 by 4 Jordan block over $\mathbf{F}_3$.

```
> K := GF(3);
> G := MatrixGroup<4,K | [1,0,0,0, 1,1,0,0, 0,1,1,0, 0,0,1,1]>;
> R := InvariantRing(G);
> P<x1,x2,x3,x4> := PolynomialRing(R);
> p := PrimaryInvariants(R);
> s := SecondaryInvariants(R);
> [TotalDegree(f): f in p];
[ 1, 2, 3, 9 ]
```

```
> [TotalDegree(f): f in s];
[ 0, 3, 4, 5, 6, 7, 8, 9 ]
> M, f := Module(R);
> M;
Full Quotient Module of degree 8
TOP Order
Column weights: 0 3 4 5 6 7 8 9
Coefficient ring:
    Graded Polynomial ring of rank 4 over GF(3)
    Lexicographical Order
    Variables: t1, t2, t3, t4
    Variable weights: 1 2 3 9
Quotient Relations:
[
    t1[7] + 2*t2[6] + t3[5],
    t1[4] + 2*t2[3] + t3[2]
]
> h := x1^5*x2 + 2*x1^3*x3^3 + 2*x2^6;
> h;
x1^5*x2 + 2*x1^3*x3^3 + 2*x2^6
> m := f(h);
> m;
t1^4*t2[1] + t1^3[2] + t2^3[1]
> // Evaluate in the primaries and secondaries:
> p[1]^4*p[2]*s[1] + p[1]^3*s[2] + p[2]^3*s[1];
x1^5*x2 + 2*x1^3*x3^3 + 2*x2^6
```

110.15 The Algebra of an Invariant Ring and Algebraic Relations

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. Suppose also that primary invariants $\{f_1, \ldots, f_n\}$ for R have been constructed, together with minimal secondary invariants $S = \{g_1, \ldots, g_m\}$ for R with respect to these primary invariants. Suppose also that the irreducible secondary invariants for R are $S = \{h_1, \ldots, h_r\}$ so that the g_i are power products of the h_i. We write $g_i = p_i(h_i)$ where the p_i are monomials of the indeterminates $t_1, \ldots, t_r$. Then R is generated as an algebra over K by the primary invariants $f_1, \ldots, f_n$ and the irreducible secondary invariants $h_1, \ldots, h_r$. Magma allows the construction of a polynomial algebra A with indeterminate names `"f1"`, `"f2"`, etc. corresponding to the primary invariants and indeterminate names `"h1"`, `"h2"`, etc. corresponding to the irreducible secondary invariants. Thus R can be regarded as an homomorphic image of A and finding the algebraic relations between these (algebra) generators of R yields a presentation of R as a quotient of a polynomial algebra. The functions in this section construct the algebra A and the algebraic relations for R. When creating the algebra A, the algebra A is assigned the print names `"f1", "f2",` "h1", "h2", etc. – the angle bracket notation or the . operator should be used to assign the variables of A to actual Magma variables.

Algebra(R)

Given an invariant ring $R = K[V]^G$, return the polynomial algebra $A = K[f_1, \ldots, f_n, h_1, \ldots, h_r]$ of which R is an homomorphic image. This function also returns a sequence Q giving the secondary invariants in terms of the irreducible secondary invariants as monomials in A. Thus $Q[i]$ is the monomial $p_i(t_i)$ mentioned in the introduction to this section. Note that the secondary invariant 1 is *not* an irreducible secondary invariant so no h-variable corresponds to it (the polynomial 1 in A simply corresponds to it).

Relations(R)

Given an invariant ring $R = K[V]^G$, return a (sorted) sequence L giving the algebraic relations amongst the algebra generators of R as elements of the algebra A corresponding to R. Thus R is isomorphic as an algebra (or ring) to the quotient of A by the ideal of A generated by the relations in L.

RelationIdeal(R)

Given an invariant ring $R = K[V]^G$, return the ideal of algebraic relations corresponding to R. This is simply the same as taking the ideal generated by the algebra A by the sequence L returned by the function Relations(R).

PrimaryAlgebra(R)

Given an invariant ring $R = K[V]^G$, return the algebra corresponding to the primary invariants of R as a graded polynomial ring (with the weights corresponding to the degrees of the primary invariants).

PrimaryIdeal(R)

Given an invariant ring $R = K[V]^G$, return the ideal generated by the primary invariants of R (this is stored in R).

Example H110E12

We create the invariant ring $R = K[V]^G$ where G is a degree-6 permutation representation of the direct product $C_3 \times C_3$ of two cyclic groups both of order 3 and K is the rational field. We construct the algebra A and the sequence Q giving the secondary invariants in terms of the irreducible secondary invariants. We then note that the degree-6 secondary invariant is obtained as the product of two degree-3 irreducible secondary invariants. We then construct the list L of algebraic relations in A for R. Thus R is isomorphic to the quotient ring $A/ < L >$. We then construct an homomorphism h from A onto R and check that the relations in L are correct. Finally, we check that the Hilbert series of the (quotient by the) ideal of A generated by L is the same as the Hilbert series of R as expected.

```
> G := PermutationGroup<6 | (1, 2, 3), (4, 5, 6)>;
> R := InvariantRing(G, RationalField());
> P := PrimaryInvariants(R);
> P;
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x4^3 + x5^3 + x6^3
]
> S := SecondaryInvariants(R);
> S;
[
    1,
    x1^2*x2 + x1*x3^2 + x2^2*x3,
    x4^2*x5 + x4*x6^2 + x5^2*x6,
    x1^2*x2*x4^2*x5 + x1^2*x2*x4*x6^2 + x1^2*x2*x5^2*x6 +
        x1*x3^2*x4^2*x5 + x1*x3^2*x4*x6^2 + x1*x3^2*x5^2*x6 +
        x2^2*x3*x4^2*x5 + x2^2*x3*x4*x6^2 + x2^2*x3*x5^2*x6
]
> H := IrreducibleSecondaryInvariants(R);
> H;
[
    x1^2*x2 + x1*x3^2 + x2^2*x3,
    x4^2*x5 + x4*x6^2 + x5^2*x6
]
> A, Q := Algebra(R);
> A;
Graded Polynomial ring of rank 8 over Rational Field
Lexicographical Order
Variables: f1, f2, f3, f4, f5, f6, h1, h2
Variable weights: 1 1 2 2 3 3 3 3
> Q;
[
    1,
    h1,
    h2,
    h1*h2
]
> // Thus S[4] must be H[1]*H[2]:
> S[4];
x1^2*x2*x4^2*x5 + x1^2*x2*x4*x6^2 + x1^2*x2*x5^2*x6 +
    x1*x3^2*x4^2*x5 + x1*x3^2*x4*x6^2 + x1*x3^2*x5^2*x6 +
    x2^2*x3*x4^2*x5 + x2^2*x3*x4*x6^2 + x2^2*x3*x5^2*x6
> H[1];
x1^2*x2 + x1*x3^2 + x2^2*x3
> H[2];
x4^2*x5 + x4*x6^2 + x5^2*x6
```

```
> H[1]*H[2] eq S[4];
true
> L := Relations(R);
> L;
[
    -1/24*f1^6 + 3/8*f1^4*f3 - 1/3*f1^3*f5 - 9/8*f1^2*f3^2 +
        2*f1*f3*f5 + f1*f3*h1 + 1/8*f3^3 - f5^2 - f5*h1 - h1^2,
    -1/24*f2^6 + 3/8*f2^4*f4 - 1/3*f2^3*f6 - 9/8*f2^2*f4^2 +
        2*f2*f4*f6 + f2*f4*h2 + 1/8*f4^3 - f6^2 - f6*h2 - h2^2
]
> // Construct homomorphism h from A onto (polynomial ring of) R:
> h := hom<A -> PolynomialRing(R) | P cat H>;
> // Check images of L under h are zero so that elements of L are relations:
> h(L);
[
    0,
    0
]
> // Create relation ideal and check its Hilbert series equals that of R:
> I := RelationIdeal(R);
> I;
Ideal of Graded Polynomial ring of rank 8 over Rational Field
Lexicographical Order
Variables: f1, f2, f3, f4, f5, f6, h1, h2
Variable weights: 1 1 2 2 3 3 3 3
Basis:
[
    f1^6 - 9*f1^4*f3 + 8*f1^3*f5 + 27*f1^2*f3^2 - 48*f1*f3*f5 -
        24*f1*f3*h1 - 3*f3^3 + 24*f5^2 + 24*f5*h1 + 24*h1^2,
    f2^6 - 9*f2^4*f4 + 8*f2^3*f6 + 27*f2^2*f4^2 - 48*f2*f4*f6 -
        24*f2*f4*h2 - 3*f4^3 + 24*f6^2 + 24*f6*h2 + 24*h2^2
]
> HilbertSeries(I);
(t^4 - 2*t^3 + 3*t^2 - 2*t + 1)/(t^10 - 4*t^9 + 6*t^8 - 6*t^7 +
    9*t^6 - 12*t^5 + 9*t^4 - 6*t^3 + 6*t^2 - 4*t + 1)
> HilbertSeries(I) eq HilbertSeries(R);
true
```

110.16 Properties of Invariant Rings

The following functions return non-trivial structural properties of invariant rings of finite groups.

```
HilbertSeries(R)
```

The Hilbert series of the invariant ring $R = K[V]^G$, returned as an element of the rational function field $\mathbf{Z}(t)$. The Molien series of G will be used if possible; otherwise (the modular matrix group case) secondary invariants for R will be constructed to determine the result.

```
HilbertSeriesApproximation(R, n)
```

The Hilbert series of the invariant ring $R = K[V]^G$, returned as a Laurent series with n known terms. The conjugacy classes of G will be used to compute the approximation.

```
IsCohenMacaulay(R)
```

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return `true` iff R is Cohen-Macaulay. This is always true in the non-modular case. Otherwise, secondary invariants for R will be constructed to determine the result.

```
FreeResolution(R)
```

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return a free resolution of (the module of) R. This is just the same as the invocation `FreeResolution(Module(R))`. The free resolution is returned as a sequence F such that $F[1]$ is M, $F[i+1]$ is the syzygy module of $F[i]$ for $i < \#F$, and the last element of F is free (its basis has no syzygies).

```
MinimalFreeResolution(R)
```

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return a minimal free resolution of (the module of) R. This is just the same as the invocation `MinimalFreeResolution(Module(R))`.

```
HomologicalDimension(R)
```

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the homological dimension of R. This is just the length of a minimal free resolution of R minus 1 (taking account of the fact that the module M of R is always included in the free resolution).

```
Depth(R)
```

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the depth of R. This is $n - d$ by the Auslander-Buchsbaum formula, where n is the rank of R and d is the homological dimension of R.

Example H110E13

We construct a minimal free resolution of the invariant ring of the group generated by the degree-5 Jordan block over $\mathbf{F}_2$ and verify that the depth is 3.

```
> K:=GF(2);
> G := MatrixGroup<5,K | [1,0,0,0,0, 1,1,0,0,0, 0,1,1,0,0,
>                         0,0,1,1,0, 0,0,0,1,1]>;
> R := InvariantRing(G);
> time F := MinimalFreeResolution(R);
Time: 0.690
> F;
Chain complex with terms of degree 3 down to -1
Dimensions of terms: 0 1 7 22 0
> Depth(R);
3
> HomologicalDimension(R);
2
```

Sections 110.17 and 110.18 present functions whose scope is not limited to the context of invariant theory.

110.17 Steenrod Operations

`SteenrodOperation(f, i)`

The i-th Steenrod operation $P^i(f)$ of f, which must be a multivariate polynomial with coefficients in a finite field, and i must be a non-negative integer.

Example H110E14

We demonstrate an elementary use of Steenrod operations.

```
> K:=GF(3);
> F4:=MatrixGroup<4,K |
>   [-1,0,0,0, 1,1,0,0, 0,0,1,0, 0,0,0,1],
>   [1,1,0,0, 0,-1,0,0, 0,1,1,0, 0,0,0,1],
>   [1,0,0,0, 0,1,-1,0, 0,0,-1,0, 0,0,1,1],
>   [1,0,0,0, 0,1,0,0, 0,0,1,1, 0,0,0,-1] >;
> R := InvariantRing(F4);
> f2 := InvariantsOfDegree(R, 2)[1];
> f4 := SteenrodOperation(f2, 1);
> f10 := SteenrodOperation(f4, 3);
> f4;
2*x1^4 + x1^3*x3 + 2*x1^3*x4 + x1*x3^3 + 2*x1*x4^3 + 2*x2^3*x3 + x2^3*x4 +
    2*x2*x3^3 + x2*x4^3 + x4^4
> f10;
2*x1^10 + x1^9*x3 + 2*x1^9*x4 + x1*x3^9 + 2*x1*x4^9 + 2*x2^9*x3 + x2^9*x4 +
    2*x2*x3^9 + x2*x4^9 + x4^10
> f4 in R;
true
> f10 in R;
true
```

110.18 Minimalization and Homogeneous Module Testing

The following functions work with collections of polynomials which are considered as generators for subalgebras or submodules of a polynomial ring. They are repeated from the chapter on multivariate polynomials since they are used extremely often in invariant theory to express an invariant in terms of the primary and secondary invariants of an invariant ring. Full descriptions of the functions are not given here. See the descriptions in the chapter on multivariate polynomials.

`MinimalAlgebraGenerators(L)`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose L is a set or sequence of k polynomials $p_1, \ldots, p_k$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by L. This function returns a minimal generating set of the algebra A as a (sorted) sequence of elements taken from L.

`HomogeneousModuleTest(P, S, F)`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose F is an element of R. This function returns whether F is in the module M (considered as a submodule of R). If the result is `true`, the function also returns a sequence $C = [c_1, \ldots, c_r]$ of length r with $c_i \in K[t_1, \ldots, t_r]$ such that $F = \sum_{i=1}^{r} c_i(p_1, \ldots, p_k) \cdot s_i$.

`HomogeneousModuleTest(P, S, L)`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose L is a sequence of length l of elements of R which are all homogeneous of (weighted) degree d. This function returns parallel sequences B and V with the following properties:

(a) B is sequence of length l of booleans such that for $1 \leq i \leq l$, $B[i]$ is `true` iff $L[i]$ is in the module M.

(b) V is a sequence of length l consisting of sequences of length r and consisting of polynomials in the polynomial ring $T = K[t_1, \ldots, t_r]$. (The polynomial ring $T = K[t_1, \ldots, t_r]$ is constructed separately but automatically with the print names `t1`, `t2`, etc.) If $B[i]$ is `false` (so $L[i]$ is not in M), $V[i]$ is a sequence of r zero polynomials. Otherwise $V[i]$ is a sequence of r polynomials $c_{i,1}, \ldots, c_{i,r}$ in T such that that $L[i] = \sum_{j=1}^{r} c_{i,j}(p_1, \ldots, p_k) \cdot s_j$.

Example H110E15

We demonstrate how the function `MinimalAlgebraGenerators` can be used to compute fundamental invariants (in fact, the Magma function `FundamentalInvariants` does just this).

```
> K := RationalField();
> G := PermutationGroup<6 | (1,2,3)(4,5,6), (1,2)(4,5)>;
> R := InvariantRing(G, K);
> P := PrimaryInvariants(R);
> P;
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x4^3 + x5^3 + x6^3
]
> S := SecondaryInvariants(R);
> S;
[
    1,
    x1*x4 + x2*x5 + x3*x6,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x1^2*x4^2 + 2*x1*x2*x4*x5 + 2*x1*x3*x4*x6 + x2^2*x5^2 + 2*x2*x3*x5*x6 +
        x3^2*x6^2,
    x1^3*x4^3 + x1^2*x2*x4*x5^2 + x1^2*x3*x4*x6^2 + x1*x2^2*x4^2*x5 +
        x1*x3^2*x4^2*x6 + x2^3*x5^3 + x2^2*x3*x5*x6^2 + x2*x3^2*x5^2*x6 +
        x3^3*x6^3
]
> MinimalAlgebraGenerators(P cat S);
[
    1,
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x1*x4 + x2*x5 + x3*x6,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3 + x4*x5*x6,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x4^3 + x5^3 + x6^3
]
```

Example H110E16

We demonstrate uses of the function `HomogeneousModuleTest` in invariant theory.

```
> // Create invariant ring R with primaries P, secondaries S
> R := InvariantRing(CyclicGroup(4), GF(2));
> P := PrimaryInvariants(R);
> S := SecondaryInvariants(R);
> #S;
5
> S[5];
x1^3*x3^2 + x1^2*x2^2*x3 + x1^2*x2*x3^2 + x1^2*x2*x4^2 +
    x1^2*x3^3 + x1^2*x3^2*x4 + x1*x2^2*x4^2 + x1*x3^2*x4^2 +
    x2^3*x4^2 + x2^2*x3^2*x4 + x2^2*x3*x4^2 + x2^2*x4^3
> // Write S[2] in terms of P and S
> HomogeneousModuleTest(P, S, S[2]^2);
true [
    t1^2*t3^2 + t2^3,
    t1*t2,
    t1^3,
    0,
    0
]
> // Find all invariants I5 of degree 5
> I5 := InvariantsOfDegree(R, 5);
> I5;
[
    x1^5 + x2^5 + x3^5 + x4^5,
    x1^4*x2 + x1*x4^4 + x2^4*x3 + x3^4*x4,
    x1^4*x3 + x1*x3^4 + x2^4*x4 + x2*x4^4,
    x1^4*x4 + x1*x2^4 + x2*x3^4 + x3*x4^4,
    x1^3*x2^2 + x1^2*x4^3 + x2^3*x3^2 + x3^3*x4^2,
    x1^3*x2*x3 + x1*x2*x4^3 + x1*x3^3*x4 + x2^3*x3*x4,
    x1^3*x2*x4 + x1*x2^3*x3 + x1*x3*x4^3 + x2*x3^3*x4,
    x1^3*x3^2 + x1^2*x3^3 + x2^3*x4^2 + x2^2*x4^3,
    x1^3*x3*x4 + x1*x2^3*x4 + x1*x2*x3^3 + x2*x3*x4^3,
    x1^3*x4^2 + x1^2*x2^3 + x2^2*x3^3 + x3^2*x4^3,
    x1^2*x2^2*x3 + x1^2*x2*x4^2 + x1*x3^2*x4^2 + x2^2*x3^2*x4,
    x1^2*x2^2*x4 + x1^2*x3*x4^2 + x1*x2^2*x3^2 + x2*x3^2*x4^2,
    x1^2*x2*x3^2 + x1^2*x3^2*x4 + x1*x2^2*x4^2 + x2^2*x3*x4^2,
    x1^2*x2*x3*x4 + x1*x2^2*x3*x4 + x1*x2*x3^2*x4 + x1*x2*x3*x4^2
]
> // Write all elements of I5 in terms of P and S
> // (the t-variables correspond to elements of P and
> // the "columns" of the inner sequences to elements of S)
> HomogeneousModuleTest(P, S, I5);
[ true, true, true, true, true, true, true, true, true, true,
true, true, true, true ]
[
    [ t1^5 + t1^3*t2 + t1^3*t3 + t1*t2^2 + t1*t3^2 + t1*t4,
        0, t1^2 + t2 + t3, 0, 0 ],
    [ t1^3*t2 + t1^3*t3 + t1*t4, t1^2 + t2, t2 + t3, 0, 0 ],
    [ t1^3*t3 + t1*t3^2 + t1*t4, 0, t1^2 + t2 + t3, 0, 0 ],
    [ t1^3*t3 + t1*t2^2 + t1*t4, t1^2 + t2, t2 + t3, 0, 0 ],
    [ t1*t2^2 + t1*t3^2, t2, t1^2, 0, 1 ],
    [ t1*t2*t3, t3, t2 + t3, 0, 1 ],
    [ t1*t2*t3 + t1*t4, 0, t1^2 + t2, 0, 0 ],
    [ t1*t3^2 + t1*t4, 0, t3, 0, 0 ],
    [ 0, t3, t3, 0, 1 ],
    [ t1*t3^2, t2, t1^2 + t2, 0, 1 ],
    [ t1*t3^2, 0, 0, 0, 1 ],
    [ t1*t3^2, 0, t2, 0, 1 ],
    [ t1*t4, 0, t3, 0, 0 ],
    [ t1*t4, 0, 0, 0, 0 ]
]
```

110.19 Attributes of Invariant Rings and Fields

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

`R‘PrimaryInvariants`

The attribute for the primary invariants of invariant ring $R = K[V]^G$. If the attribute `R‘PrimaryInvariants` is examined, either the current primary invariants of R are set so they are returned or an error results. If the attribute `R‘PrimaryInvariants` is set by assignment, it must be sequence of n algebraically-independent invariants of G, where n is the rank of R. Magma will not necessarily check that this condition is met since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value. Note that this attribute is useful when it is desired to compute secondary invariants of R with respect to some specially constructed primary invariants which would not be constructed by the automatic algorithm in Magma.

`R‘SecondaryInvariants`

The attribute for the secondary invariants of invariant ring $R = K[V]^G$. If the attribute R‘SecondaryInvariants is examined, either the current primary invariants of R are set so they are returned or an error results. If the attribute R‘SecondaryInvariants is set by assignment to Q, primary invariants for R must already be defined, and Q must be sequence of secondary invariants with respect to the primary invariants of R. Magma will not necessarily check that this condition is met since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value.

`R‘HilbertSeries`

The attribute for the Hilbert series of invariant ring $R = K[V]^G$. If the attribute R‘HilbertSeries is examined, either the Hilbert series of R is computed so it is returned or an error results. If the attribute R‘HilbertSeries is set by assignment to H, H must be rational function in the function field $Z(t)$ and equal to the Hilbert series of R. Magma will not necessarily check that this condition is met since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value.

Example H110E17

We demonstrate elementary uses of attributes.

```
> // Create group G and subgroup H of G and invariant rings
> // RG and RH of G and H respectively.
> G := CyclicGroup(4);
> H := sub<G|G.1^2>;
> RG := InvariantRing(G, GF(2));
> RH := InvariantRing(H, GF(2));
>
> // Create Hilbert Series S of RG and set it in RG.
> F<t> := FunctionField(IntegerRing());
> S := (t^3 + t^2 - t + 1)/(t^8 - 2*t^7 + 2*t^5 - 2*t^4 +
>     2*t^3 - 2*t + 1);
> RG‘HilbertSeries := S;
>
> // Note RG has no primary invariants yet so let Magma compute them as PG.
>  RG‘PrimaryInvariants;
>> RG‘PrimaryInvariants;
     ^
Runtime error in ‘: Attribute ’PrimaryInvariants’ for this structure is valid
but not assigned
> PG := PrimaryInvariants(RG);
> PG;
[
    x1 + x2 + x3 + x4,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4,
    x1*x2*x3*x4
]
>
> // Set primary invariants of RH to PG and compute secondary
> // invariants of RH with respect to PG.
> RH`PrimaryInvariants := PG;
> SecondaryInvariants(RH);
[
    1,
    x2 + x4,
    x2*x4,
    x1*x2 + x1*x3 + x2^2 + x2*x4 + x3*x4 + x4^2,
    x1^2*x2 + x1*x2*x3 + x1*x3*x4 + x2^3 + x3^2*x4 + x4^3,
    x1^2*x2 + x1*x2^2 + x1*x2*x3 + x1*x2*x4 + x1*x3*x4 + x1*x4^2
        + x2^3 + x2^2*x3 + x2*x3*x4 + x3^2*x4 + x3*x4^2 + x4^3,
    x1*x2*x4^2 + x2^2*x3*x4 + x2^2*x4^2,
    x1^2*x2*x4^2 + x1*x2^2*x3*x4 + x1*x2^2*x4^2 + x1*x2*x3*x4^2 +
        x2^3*x4^2 + x2^2*x3^2*x4 + x2^2*x3*x4^2 + x2^2*x4^3
]
```

110.20 Invariant Rings of Linear Algebraic Groups

By definition, a linear algebraic group is an affine variety G together with morphisms giving G the structure of a group. In the invariant theory algorithms of Magma, the group structure of G is nowhere required. Therefore an algebraic group will be defined by simply giving polynomials defining G as an affine variety. A G-module of an algebraic group is a finite dimensional vector space K^n together with a morphism $G \to \mathrm{GL}_n$ of algebraic groups. In Magma, such a morphism is given by specifying an n by n matrix whose entries are polynomials in the same variables as the polynomials specifying G (for more details, see section 110.6). An invariant ring of a linear algebraic group is constructed by giving a linear algebraic group together with a G-module. Magma makes no checks that the variety defined by the user has a multiplication making it into an algebraic group, or that the morphism $G \to \mathrm{GL}_n(K)$ really provides an action of G. If they do not, the computations will have unpredictable results. Likewise, Magma is unable to decide whether an algebraic group is reductive or linearly reductive. Therefore the user should indicate whether a group has these properties at creation by the options described below. This is important because Derksen's algorithm only works for linearly reductive groups.

110.20.1 Creation

`InvariantRing(I, A)`

`Reductive`	BoolElt	*Default :* `false`
`LinearlyReductive`	BoolElt	*Default :* `false`
`PolynomialRing`	RngMPol	*Default :*

Construct the invariant ring R for the algebraic group G defined by the ideal I and representation matrix A.

If the parameter `Reductive` is set to `true`, then G is assumed to be reductive, while if the parameter `LinearlyReductive` is set to `true`, then G is assumed to be linearly reductive.

If the parameter `PolynomialRing` is set to a value P, then P is used as the polynomial ring in which the invariants of R will lie.

`BinaryForms(N, p)`

`BinaryForms(n, p)`

Let $N = [n_1, \ldots, n_k]$ be a sequence of positive integers and let p be a positive prime or zero. Let $G = \mathrm{SL}_2(K)$ with K an algebraically closed field of characteristic p. This function defines the action on a direct sum of spaces of binary forms with degrees given by the n_i. The function returns three items: the ideal I_G defining G as an algebraic group, the representation matrix A (as a sequence of sequences of polynomials), and a polynomial ring on which G acts with appropriate naming of variables.

The second version of the function is given an integer n, and takes N to be $[n]$.

110.20.2 Access

`GroupIdeal(R)`

Given an invariant ring R defined over an algebraic group G, return the ideal I defining G.

`Representation(R)`

Given an invariant ring R defined over an algebraic group G, return the representation matrix A for G.

110.20.3 Functions

`InvariantsOfDegree(R, d)`

Return a K-basis of the space R_d of the homogeneous invariants of degree d in the invariant ring $R = K[V]^G$ of the algebraic group G over the field K as a sequence of polynomials.

`FundamentalInvariants(R)`

`Optimize`	BoolElt	*Default :* `true`
`Minimize`	BoolElt	*Default :* `true`
`MinimizeHilbert`	BoolElt	*Default :* `true`
`Force`	BoolElt	*Default :* `false`

Given an invariant ring R defined over an algebraic group, return a sequence of fundamental invariants of R, using Derksen's algorithm.

By default, the computation of homogeneous invariants is optimized by extending at each the degree the basis obtained from multiplying lower-degree invariants by appropriate monomials. This method can be suppressed by setting `Optimize` to `false`. By default the generators will be minimal. By setting the parameter `Minimize` to `false`, no minimization will be attempted. By setting the parameter `MinimizeHilbert` to `false`, the basis of the Hilbert ideal will not be minimized.

By default the group must be linearly reductive. Setting the parameter `Force` to `true` will force the application of Derksen's algorithm even though the group may not be linearly reductive.

`DerksenIdeal(R)`

Given an invariant ring R defined over an algebraic group, return a sequence of generators of the Derksen ideal of R. The Derksen ideal is an ideal D of $P[y_1, \ldots, y_n]$, where $P = K[x_1, \ldots, x_n]$ is the ambient polynomial ring of R, and the y_i are new indeterminates. By definition, D is the intersection of all the ideals

$$\langle y_1 - g(x_1), \ldots, y_n - g(x_n) \rangle$$

for all $g \in G$, the group of R. Geometrically, D is the vanishing ideal of the subset

$$\{(x, g(x)) | x \in K^n, g \in G\}$$

of the cartesian product $K^n \times K^n$.

`HilbertIdeal(R)`

`Minimize`	BoolElt	*Default :* `true`
`Force`	BoolElt	*Default :* `false`

Given an invariant ring R defined over a linear algebraic group, return the Hilbert ideal of R. This is the ideal in the polynomial ring generated by all non-constant, homogeneous invariants. The result is a sequence of homogeneous generators (not necessarily invariant).

By default the generators will be minimal. By setting the parameter `Minimize` to `false`, no minimization will be attempted. Also, setting the parameter `Force` to `true` will force the application of Derksen's algorithm even though the group may not be linearly reductive.

Example H110E18

We consider invariant ring of the group $G = SL_2(\mathbf{Q})$, which is is characterised by the equation $\det(A) = 1$.

```
> Q := RationalField();
> P<[a]>:=PolynomialRing(Q, 4);
> A := MatrixRing(P,2)!a;
> IG := ideal<P | Determinant(A) - 1>;
> IG;
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: a[1], a[2], a[3], a[4]
Basis:
[
    a[1]*a[4] - a[2]*a[3] - 1
]
```

The simultaneous action of G on three vectors is given by the matrix $I_3 \otimes A$:

```
> T := TensorProduct(MatrixRing(P, 3) ! 1, A);
> T;
[a[1] a[2]    0    0    0    0]
[a[3] a[4]    0    0    0    0]
[   0    0 a[1] a[2]    0    0]
[   0    0 a[3] a[4]    0    0]
[   0    0    0    0 a[1] a[2]]
[   0    0    0    0 a[3] a[4]]
```

We create the invariant ring R of G (which is reductive) with this action and compute fundamental invariants.

```
> IR := InvariantRing(IG, T: Reductive);
> FundamentalInvariants(IR);
[
    x3*x6 - x4*x5,
    x1*x6 - x2*x5,
    x1*x4 - x2*x3
]
```

We see that there are three fundamental invariants. It is well known that the invariant ring of the simultaneous action of SL_n on m vectors is generated by the minors of the $n \times m$ matrix formed by the vectors. We can see this in the present case.

```
> R<x1,x2,x3,x4,x5,x6> := PolynomialRing(Q, 6);
> M := Matrix([[x1,x3,x5], [x2,x4,x6]]);
> M;
[x1 x3 x5]
[x2 x4 x6]
> Minors(M, 2);
[
    x1*x4 - x2*x3,
    -x1*x6 + x2*x5,
    x3*x6 - x4*x5
]
```

Example H110E19

As a second example, we consider the representation of the group $SL_2(\mathbf{Q}) \times SL_2(\mathbf{Q}) \times SL_2(\mathbf{Q})$ given by the tensor product of the canonical representation:

```
> n:=3;
> P<[x]>:=PolynomialRing(RationalField(), n*4, "grevlex");
> L_A := [MatrixRing(P,2)!x[i..i+3]:i in [1..n*4 by 4]];
> IG := ideal<P|[Determinant(A)-1:A in L_A]>;
> IG;
Ideal of Polynomial ring of rank 12 over Rational Field
Graded Reverse Lexicographical Order
Variables: x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8],
x[9], x[10], x[11], x[12]
Basis:
[
    -x[2]*x[3] + x[1]*x[4] - 1,
    -x[6]*x[7] + x[5]*x[8] - 1,
    -x[10]*x[11] + x[9]*x[12] - 1
]
>
> M:=L_A[1];
> for i:=2 to n do
>     M:=TensorProduct(M,L_A[i]);
> end for;
> M;
[x[1]*x[5]*x[9]   x[1]*x[5]*x[10]   x[1]*x[6]*x[9]   x[1]*x[6]*x[10]
    x[2]*x[5]*x[9]   x[2]*x[5]*x[10]   x[2]*x[6]*x[9]   x[2]*x[6]*x[10]]
[x[1]*x[5]*x[11]   x[1]*x[5]*x[12]   x[1]*x[6]*x[11]   x[1]*x[6]*x[12]
    x[2]*x[5]*x[11]   x[2]*x[5]*x[12]   x[2]*x[6]*x[11]   x[2]*x[6]*x[12]]
[x[1]*x[7]*x[9]   x[1]*x[7]*x[10]   x[1]*x[8]*x[9]   x[1]*x[8]*x[10]
    x[2]*x[7]*x[9]   x[2]*x[7]*x[10]   x[2]*x[8]*x[9]   x[2]*x[8]*x[10]]
[x[1]*x[7]*x[11]   x[1]*x[7]*x[12]   x[1]*x[8]*x[11]   x[1]*x[8]*x[12]
    x[2]*x[7]*x[11]   x[2]*x[7]*x[12]   x[2]*x[8]*x[11]   x[2]*x[8]*x[12]]
[x[3]*x[5]*x[9]   x[3]*x[5]*x[10]   x[3]*x[6]*x[9]   x[3]*x[6]*x[10]
    x[4]*x[5]*x[9]   x[4]*x[5]*x[10]   x[4]*x[6]*x[9]   x[4]*x[6]*x[10]]
[x[3]*x[5]*x[11]   x[3]*x[5]*x[12]   x[3]*x[6]*x[11]   x[3]*x[6]*x[12]
    x[4]*x[5]*x[11]   x[4]*x[5]*x[12]   x[4]*x[6]*x[11]   x[4]*x[6]*x[12]]
[x[3]*x[7]*x[9]   x[3]*x[7]*x[10]   x[3]*x[8]*x[9]   x[3]*x[8]*x[10]
    x[4]*x[7]*x[9]   x[4]*x[7]*x[10]   x[4]*x[8]*x[9]   x[4]*x[8]*x[10]]
[x[3]*x[7]*x[11]   x[3]*x[7]*x[12]   x[3]*x[8]*x[11]   x[3]*x[8]*x[12]
    x[4]*x[7]*x[11]   x[4]*x[7]*x[12]   x[4]*x[8]*x[11]   x[4]*x[8]*x[12]]
> IR:=InvariantRing(IG, M: Reductive);
> time FundamentalInvariants(IR);
[
    x1^2*x8^2 - 2*x1*x2*x7*x8 - 2*x1*x3*x6*x8 - 2*x1*x4*x5*x8 +
        4*x1*x4*x6*x7 + x2^2*x7^2 + 4*x2*x3*x5*x8 - 2*x2*x3*x6*x7 -
        2*x2*x4*x5*x7 + x3^2*x6^2 - 2*x3*x4*x5*x6 + x4^2*x5^2
]
Time: 0.610
> time DerksenIdeal(IR);
[
    y1^2*y8^2 - 2*y1*y2*y7*y8 - 2*y1*y3*y6*y8 - 2*y1*y4*y5*y8 + 4*y1*y4*y6*y7 +
        y2^2*y7^2 + 4*y2*y3*y5*y8 - 2*y2*y3*y6*y7 - 2*y2*y4*y5*y7 + y3^2*y6^2 -
        2*y3*y4*y5*y6 + y4^2*y5^2 - x1^2*x8^2 + 2*x1*x2*x7*x8 + 2*x1*x3*x6*x8 +
        2*x1*x4*x5*x8 - 4*x1*x4*x6*x7 - x2^2*x7^2 - 4*x2*x3*x5*x8 +
        2*x2*x3*x6*x7 + 2*x2*x4*x5*x7 - x3^2*x6^2 + 2*x3*x4*x5*x6 - x4^2*x5^2
]
Time: 0.010
> time HilbertIdeal(IR);
[
    x1^2*x8^2 - 2*x1*x2*x7*x8 - 2*x1*x3*x6*x8 - 2*x1*x4*x5*x8 + 4*x1*x4*x6*x7 +
        x2^2*x7^2 + 4*x2*x3*x5*x8 - 2*x2*x3*x6*x7 - 2*x2*x4*x5*x7 + x3^2*x6^2 -
        2*x3*x4*x5*x6 + x4^2*x5^2
]
Time: 0.000
```

So in this case, we find that the invariant ring is generated by a single polynomial.

Example H110E20

We compute fundamental invariants for the invariant ring of $G = \mathrm{SL}_2(\mathbf{Q})$ acting on a space of binary forms.

```
> IG, A := BinaryForms([1,1,2,2], 0);
> IG;
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: t1, t2, t3, t4
Basis:
[
    t1*t4 - t2*t3 - 1
]
> A;
[t4   -t3   0   0   0   0   0   0   0   0]
[-t2   t1   0   0   0   0   0   0   0   0]
[0   0   t4   -t3   0   0   0   0   0   0]
[0   0   -t2   t1   0   0   0   0   0   0]
[0   0   0   0   t4^2   -t3*t4   t3^2   0   0   0]
[0   0   0   0   -2*t2*t4   t1*t4 + t2*t3   -2*t1*t3   0   0   0]
[0   0   0   0   t2^2   -t1*t2   t1^2   0   0   0]
[0   0   0   0   0   0   0   t4^2   -t3*t4   t3^2]
```

```
[0   0   0   0   0   0   0   -2*t2*t4   t1*t4 + t2*t3   -2*t1*t3]
[0   0   0   0   0   0   0   t2^2   -t1*t2   t1^2]
> R:=InvariantRing(IG,A: LinearlyReductive);
> time FundamentalInvariants(R);
[
    x8*x10 - 1/4*x9^2,
    x5*x10 - 1/2*x6*x9 + x7*x8,
    x5*x7 - 1/4*x6^2,
    x1*x4 - x2*x3,
    x1*x3*x7 - 1/2*x1*x4*x6 - 1/2*x2*x3*x6 + x2*x4*x5,
    x1*x3*x10 - 1/2*x1*x4*x9 - 1/2*x2*x3*x9 + x2*x4*x8,
    x3^2*x10 - x3*x4*x9 + x4^2*x8,
    x3^2*x7 - x3*x4*x6 + x4^2*x5,
    x1^2*x10 - x1*x2*x9 + x2^2*x8,
    x1^2*x7 - x1*x2*x6 + x2^2*x5,
    x1*x2*x5*x10 - x1*x2*x7*x8 - x2^2*x5*x9 + x2^2*x6*x8,
    x1*x4*x5*x10 - 1/2*x1*x4*x6*x9 + x1*x4*x7*x8 - 1/2*x2*x3*x5*x10 +
        1/2*x2*x3*x6*x9 - 3/2*x2*x3*x7*x8 - 1/2*x2*x4*x5*x9 + 1/2*x2*x4*x6*x8,
    x3*x4*x5*x10 - x3*x4*x7*x8 - x4^2*x5*x9 + x4^2*x6*x8
]
Time: 0.650
```

Example H110E21

We do simple computations on an invariant ring of an algebraic group. The group is not reductive, so fundamental invariants cannot be computed, but invariants of specific degrees can be.

```
> K := RationalField();
> Pa<a,b> := PolynomialRing(K, 2);
> IG := ideal<Pa|>;
> A := Matrix(7,
> [1, 0, 0, 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
> 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
> a, 1, 0, 0, 0, 0, 0, b, 0, 1 ]);
> A;
[1 0 0 0 0 0 0]
[a 1 0 0 0 0 0]
[0 0 1 0 0 0 0]
[0 0 a 1 0 0 0]
[0 0 0 0 1 0 0]
[0 0 0 0 a 1 0]
[0 0 0 0 b 0 1]
> R:=InvariantRing(IG, A);
> R;
Invariant Ring of algebraic group
Field of definition:
    Rational Field
> InvariantsOfDegree(R, 1);
```

```
[
    x1,
    x3,
    x5
]
> InvariantsOfDegree(R, 2);
[
    x1^2,
    x1*x3,
    x1*x4 - x2*x3,
    x1*x5,
    x1*x6 - x2*x5,
    x3^2,
    x3*x5,
    x3*x6 - x4*x5,
    x5^2
]
>  FundamentalInvariants(R);
>> FundamentalInvariants(R);
                         ^
Runtime error in 'FundamentalInvariants': Computing fundamental invariants (via
Derksen's algorithm) is only possible for linearly reductive groups
```

110.21 Invariant Fields

If G is a group acting on a polynomial ring $K[x_1, \ldots, x_n]$, it also acts on the rational function field $K(x_1, \ldots, x_n)$ by homomorphic extension. The invariants field $K(x_1, \ldots, x_n)^G$ is the field consisting of all functions which are fixed by G. Magma allows the construction of the invariant field by the function `InvariantField`. All that was said above about possible arguments of `InvariantRing` and access functions for invariant rings carries over to invariant fields. The category of invariant fields is `FldInvar`.

110.21.1 Creation

```
InvariantField(G, K)
InvariantField(G)
InvariantField(I, A)
```

Reductive	BoolElt	*Default :* false
LinearlyReductive	BoolElt	*Default :* false
FunctionField	FldFunRat	*Default :*

Create the invariant field for the group G over the field K. The arguments and parameters are the same as for the function `InvariantRing`, in the three cases of permutation groups, matrix groups, and algebraic groups.

110.21.2 Access

`FunctionField(F)`

Given an invariant field F, return the underlying function field of F.

`Group(F)`

Given an invariant field F, return the underlying group of F.

`GroupIdeal(F)`

Given an invariant field F defined over an algebraic group G, return the ideal I defining G.

`Representation(F)`

Given an invariant field F defined over an algebraic group G, return the representation matrix A for G.

110.21.3 Functions for Invariant Fields

This section describes functions that apply to invariant fields.

`FundamentalInvariants(F)`

`Al`	MonStgElt	*Default :* "*BethMuellerQuade*"
`Minimize`	BoolElt	*Default :* `true`
`Min`	RngIntElt	*Default :* 0
`BottomUpTo`	RngIntElt	*Default :* 0

Given an invariant field F, return a sequence of fundamental invariants of F which generate F as an algebra over the base field of the ambient rational function field of F.

By default this function uses the algorithm of Beth and Müller-Quade [MQB99]. By setting the parameter `Al` to `"FleischmannKemperWoodcock"`, an alternative algorithm of Fleischmann, Kemper and Woodcock will be used.

By default the returned invariants will be minimal (in the sense of 'non-redundant'). By setting the parameter `Minimize` to `false`, no minimization will be attempted. The other parameters apply to the minimization and are as in the function `MinimizeGenerators` below.

`DerksenIdeal(F)`

Given an invariant field F, return the Derksen ideal of F. This is an ideal D in $K[y_1 \ldots y_n]$, where $K = k(x_1 \ldots x_n)$ is the ambient rational function field of F, and the y_i are new indeterminates. By definition, D is the intersection of all the ideals

$$< y_1 - g(x_1), \ldots, y_n - g(x_n) >$$

for $g \in G$, the group of R. The function returns D as an ideal with a Groebner basis.

```
MinimizeGenerators(L)
```

Min RngIntElt *Default :* 0

BottomUpTo RngIntElt *Default :* 0

Suppose L is a set or sequence of non-constant elements of a rational function field. This function selects a minimal (in the sense of 'irredundant') subset of L which generates the same subfield as L. The function returns a sequence of such minimal generators.

If the parameter `Min` is set to $m > 0$, then the function stops when a generating set with m elements is reached ($m = 0$ is the default and implies no limit).

If the parameter `BottomUpTo` is set to $b > 0$, then the function first tries to eliminate generators by testing if they lie in the subfield generated by a small number of elements from L. This small number is limited by b.

```
QuadeIdeal(L)
```

Fy BoolElt *Default :*

LargeIdeal BoolElt *Default :* `false`

Suppose L is a non-empty set or sequence of non-constant elements from a rational function field $F = k(x_1, \ldots, x_n)$, generating a subfield $K = k(L)$. The Quade ideal, introduced in [MQS99], is the ideal in $F[y_1, \ldots, y_n]$ generated by the kernel of the map $K[y_1, \ldots, y_n] \to F$ given by $y_i \mapsto x_i$. This function returns the Quade ideal (with its basis being a Groebner basis).

The parameter `Fy` may be set to a polynomial ring P of rank n over F, so that the result is an ideal of P. If the parameter `LargeIdeal` is set to `true`, then an ideal in a larger polynomial ring is returned, whose intersection with $F[y_1, \ldots, y_n]$ is the Quade ideal.

Example H110E22

This example works with the invariant field of the finite group C_3 over the rational field.

```
> IF := InvariantField(CyclicGroup(3), RationalField());
> time L := FundamentalInvariants(IF);
Time: 1.780
> L;
[
    x1 + x2 + x3,
    (x1^2*x2 - 3*x1*x2*x3 + x1*x3^2 + x2^2*x3)/(x1^2 - x1*x2 - x1*x3 + x2^2 -
        x2*x3 + x3^2),
    (x1^3 - x1^2*x3 - x1*x2^2 + x2^3 - x2*x3^2 + x3^3)/(x1^2 - x1*x2 - x1*x3 +
        x2^2 - x2*x3 + x3^2)
]
> time DerksenIdeal(IF);
Ideal of Polynomial ring of rank 3 over Multivariate rational function field of
    rank 3 over Rational Field
Graded Reverse Lexicographical Order
```

```
Variables: y1, y2, y3
Dimension 0
Groebner basis:
[
    y2^2 + (-x1^3 + x1^2*x3 + x1*x2^2 - x2^3 + x2*x3^2 - x3^3)/(x1^2 - x1*x2 -
        x1*x3 + x2^2 - x2*x3 + x3^2)*y2 + (-x1^2*x2 + x1^2*x3 + x1*x2^2 -
        x1*x3^2 - x2^2*x3 + x2*x3^2)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 +
        x3^2)*y3 + (x1^3*x2 - x1^2*x2^2 - x1^2*x3^2 + x1*x3^3 + x2^3*x3 -
        x2^2*x3^2)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2),
    y2*y3 + (-x1^2*x2 + 3*x1*x2*x3 - x1*x3^2 - x2^2*x3)/(x1^2 - x1*x2 - x1*x3 +
        x2^2 - x2*x3 + x3^2)*y2 + (-x1^2*x3 - x1*x2^2 + 3*x1*x2*x3 -
        x2*x3^2)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2)*y3 + (x1^2*x2^2 -
        x1^2*x2*x3 + x1^2*x3^2 - x1*x2^2*x3 - x1*x2*x3^2 + x2^2*x3^2)/(x1^2 -
        x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2),
    y3^2 + (x1^2*x2 - x1^2*x3 - x1*x2^2 + x1*x3^2 + x2^2*x3 - x2*x3^2)/(x1^2 -
        x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2)*y2 + (-x1^3 + x1^2*x2 + x1*x3^2 -
        x2^3 + x2^2*x3 - x3^3)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2)*y3 +
        (x1^3*x3 - x1^2*x2^2 - x1^2*x3^2 + x1*x2^3 - x2^2*x3^2 + x2*x3^3)/(x1^2
        - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2),
    y1 + y2 + y3 - x1 - x2 - x3
]
```

Example H110E23

We can compute with the invariant field of the non-reductive group presented above.

```
> K := RationalField();
> Pa<a,b> := PolynomialRing(K, 2);
> IG := ideal<Pa|>;
> A := Matrix(7,
> [1, 0, 0, 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
> 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
> a, 1, 0, 0, 0, 0, 0, b, 0, 1 ]);
> A;
[1 0 0 0 0 0 0]
[a 1 0 0 0 0 0]
[0 0 1 0 0 0 0]
[0 0 a 1 0 0 0]
[0 0 0 0 1 0 0]
[0 0 0 0 a 1 0]
[0 0 0 0 b 0 1]
> IF := InvariantField(IG, A);
> IF;
Invariant field of algebraic group
Field of definition: Rational Field
> time FundamentalInvariants(IF);
[
    x5,
    x3/x5,
    x1,
    (x1*x6 - x2*x5)/x5,
    (x3*x6 - x4*x5)/x5
]
Time: 0.010
> DerksenIdeal(IF);
Ideal of Polynomial ring of rank 7 over Multivariate rational function field of
    rank 7 over Rational Field
Graded Reverse Lexicographical Order
Variables: y1, y2, y3, y4, y5, y6, y7
Groebner basis:
[
    y1 - x1,
    y2 - x1/x5*y6 + (x1*x6 - x2*x5)/x5,
    y3 - x3,
    y4 - x3/x5*y6 + (x3*x6 - x4*x5)/x5,
    y5 - x5
]
```

110.22 Invariants of the Symmetric Group

Magma includes basic functions for working with symmetric polynomials, which are invariants of the symmetric group.

`ElementarySymmetricPolynomial(P, k)`

Given a polynomial ring P of rank n, and an integer k with $1 \leq k \leq n$, return the k-th elementary symmetric polynomial of P.

`IsSymmetric(f)`

`IsSymmetric(f, S)`

Given a polynomial f from a polynomial ring P of rank n, return whether f is a symmetric polynomial of P (i.e., is symmetric in all the n variables of P). If the answer is true, a polynomial g from a new polynomial ring of rank n is returned such that $f = g(e_1, \ldots, e_n)$, where e_i is the i-th elementary symmetric polynomial of P. If g is desired to be a member of a particular polynomial ring S of rank n (to obtain predetermined names of variables, for example), then S may also be passed.

Example H110E24

We create a symmetric polynomial from $\mathbf{Q}[a, b, c, d]$ and express it in terms of the elementary symmetric polynomials.

```
> P<a, b, c, d> := PolynomialRing(RationalField(), 4, "grevlex");
> f :=
> a^2*b^2*c*d + a^2*b*c^2*d + a*b^2*c^2*d + a^2*b*c*d^2 + a*b^2*c*d^2 +
>     a*b*c^2*d^2 - a^2*b^2*c - a^2*b*c^2 - a*b^2*c^2 - a^2*b^2*d -
>     3*a^2*b*c*d - 3*a*b^2*c*d - a^2*c^2*d - 3*a*b*c^2*d - b^2*c^2*d -
>     a^2*b*d^2 - a*b^2*d^2 - a^2*c*d^2 - 3*a*b*c*d^2 - b^2*c*d^2 -
>     a*c^2*d^2 - b*c^2*d^2 + a + b + c + d;
> // Check orbit under Sym(4) has size one:
> #(f^Sym(4));
1
> Q<e1, e2, e3, e4> := PolynomialRing(RationalField(), 4);
> l, E := IsSymmetric(f, Q);
> l;
true
> E;
e1 - e2*e3 + e2*e4
```

In the following example, we use a rational function field to define parameters a and b which occur as coefficients of the symmetric polynomial f.

```
> F<a,b> := FunctionField(RationalField(), 2);
> P<x1,x2,x3,x4,x5> := PolynomialRing(F, 5, "grevlex");
> y1 := x1^4 + x1^2*a + x1*b;
> y2 := x2^4 + x2^2*a + x2*b;
> y3 := x3^4 + x3^2*a + x3*b;
> y4 := x4^4 + x4^2*a + x4*b;
> y5 := x5^4 + x5^2*a + x5*b;
> f := y1*y2 + y1*y3 + y1*y4 + y1*y5 + y2*y3 + y2*y4 +
>      y2*y5 + y3*y4 + y3*y5 + y4*y5;
> Q<e1,e2,e3,e4,e5> := PolynomialRing(F,  5);
> l,E := IsSymmetric(f, Q);
> l, E;
true b*e1^3*e2 - 2*a*e1^3*e3 - 4*e1^3*e5 + a*e1^2*e2^2 +
    4*e1^2*e2*e4 + 2*e1^2*e3^2 - b*e1^2*e3 + 2*a*e1^2*e4 -
    4*e1*e2^2*e3 - 3*b*e1*e2^2 + 4*a*e1*e2*e3 + 8*e1*e2*e5 +
    a*b*e1*e2 - 8*e1*e3*e4 - 2*a^2*e1*e3 + b*e1*e4 - 6*a*e1*e5 +
    e2^4 - 2*a*e2^3 - 4*e2^2*e4 + a^2*e2^2 + 4*e2*e3^2 +
    5*b*e2*e3 + 2*a*e2*e4 + b^2*e2 - 3*a*e3^2 - 4*e3*e5 -
    3*a*b*e3 + 6*e4^2 + 2*a^2*e4 - 5*b*e5
```

110.23 Bibliography

[**AM94**] A. Adem and R.J. Milgram. *Cohomology of Finite Groups.* Grundlehren der Mathematischen Wissenschaften. Springer, Berlin-New York-Heidelberg, 1994.

[**Der99**] Harm Derksen. Computation of Invariants for Reductive Groups. *Adv. Math.*, 141:366–384, 1999.

[**Kem96**] Gregor Kemper. Calculating Invariant Rings of Finite Groups over Arbitrary Fields. *J. Symbolic Comp.*, 21(3):351–366, 1996.

[**Kem99**] Gregor Kemper. An Algorithm to Calculate Optimal Homogeneous Systems of Parameters. *J. Symbolic Comp.*, 27(2):171–184, 1999.

[**Kin07**] Simon King. Minimal generating sets of non-modular invariant rings of finite groups. URL:http://arxiv.org/abs/math/0703035, 2007.

[**KS97**] Gregor Kemper and Allan Steel. Some Algorithms in Invariant Theory of Finite Groups. In P. Dräxler, G.O. Michler, and C.M. Ringel, editors, *Computational Methods for Representations of Groups and Algebras, Euroconference in Essen, April 1-5 1997*, number 173 in Progress in Mathematics, Basel, 1997. Birkhäuser.

[**MQB99**] Jörg Müller-Quade and Thomas Beth. Calculating Generators for Invariant Fields of Linear Algebraic Groups. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (Honolulu, HI, 1999)*, number 1719 in LNCS, pages 392–403, Berlin, 1999. Springer.

[**MQS99**] Jörg Müller-Quade and Rainer Steinwandt. Basic algorithms for rational function fields. *J. Symbolic Comp.*, 27(2):143–170, 1999.

111 DIFFERENTIAL RINGS

Chapter 111
DIFFERENTIAL RINGS

111.1 Introduction

The Galois theory of linear differential equations, or differential Galois theory, is the analogue of the classical Galois theory of polynomials for linear differential equations. Generally speaking one studies linear differential equations, that is differential equations of the form

$$L(y) = a_n y^{(n)} + a_{n-1} y^{(n-1)} + \cdots + a_1 y^{(1)} + a_0 y = 0,$$

in which the coefficients a_i are contained in some ring. The natural analogue of a field in the classical case is the notion of a differential field, that is a specific case of a differential ring. A *differential ring* F is equipped with an additive map $\delta_F : F \to F$ called a *derivation*, satisfying the multiplicative rule

$$\delta_F(a \cdot b) = \delta_F(a) \cdot b + a \cdot \delta_F(b), \quad a, b \in F.$$

A classical derivation is the usual derivative. All differential rings have a ring structure and have a map defined on them. A differential ring that is also a field is called a *differential field*.

The differential rings have type `RngDiff` and their elements have type `RngDiffElt`. All differential rings contain a *differential ring of constants* on which the derivation acts as the zero map. The differential rings and their elements inherit all functionality of the rings from which the differential ring is created. We call the ring from which a differential ring F is created the *underlying ring* of F.

A solution of a differential equation is an element of some differential field. It can happen that a solution is not an element of a given differential field F, but is an element of a differential extension of F. By this we mean a differential field (ring) M with $F \subset M$ such that the derivations satisfy $\delta_M|_F = \delta_F$. This is completely analogous to field extensions induced by solutions of a polynomial.

To clearly describe linear differential equations in Magma we formalize the concept of taking the derivative. To a differential field F with derivation δ_F, one associates a non–commutative ring $F[D]$, *the ring of linear differential operators.* An element of $F[D]$ is called a *differential operator.* A differential operator of *degree* $n \in \mathbf{Z}_{\geq 0}$ in $F[D]$ is of the form

$$L = a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0,$$

with $a_n \neq 0$ and all $a_i \in F$. Addition in $F[D]$ is term–wise and the multiplication of elements in $F[D]$ is determined by the rule

$$D * a = aD + \delta_F(a), \quad a \in F.$$

With these concepts $L(y) = 0$ is the linear differential equation

$$a_n\delta_F^n(y) + a_{n-1}\delta_F^{n-1}(y) + \cdots + a_1\delta_F(y) + a_0 y = 0.$$

For an introduction to the basic concepts in differential Galois theory, one is encouraged to consult [vdPS03]. This book is used as the basis for the implementation of differential rings, fields and operator rings in Magma.

111.2 Differential Rings and Fields

111.2.1 Creation

There are two ways to create a differential ring. The first creation is a general creation of a differential ring, for which the user specifies the ring and its derivation. The second creates a differential field which has the structure of a rational function field of transcendence degree 1 over its base field. Its derivation is specified by a differential.

Once a differential ring is created one can ask for its ring or field of fractions.

`DifferentialRing(P, f, C)`

Given a ring P and derivation f acting on P, return the differential ring isomorphic to P, with induced derivation f acting on it, and ring of constants C. The ring C should be a subring of P on which f is zero.

Example H111E1

Here we illustrate the creation and printing of a general differential ring.

```
> P := PolynomialRing(Rationals());
> f := map<P->P | a:->5*Derivative(a)>;
> R := DifferentialRing(P, f, Rationals());
> R;
Differential Ring of Univariate Polynomial Ring over
Rational Field with derivation given by Mapping
from: RngUPol: P to RngUPol: P given by a rule [no inverse]
```

`RationalDifferentialField(C)`

The differential field in one variable over the constant field C. If this field is called F, say, then the derivation on F is given by $d/(1)d(F.1)$, where $F.1$ is the variable of F, and $(1)d(F.1)$ is its differential in the differential space of F. Any exact field with polynomial GCD is valid input for C.

Example H111E2

Here we illustrate the creation and printing of the differential field obtained from the command `RationalDifferentialField`.

```
> F<z> := RationalDifferentialField(Rationals());
> F;
Differential Ring of Algebraic function field defined over
Rational Field by $.2 - 4711 with
derivation given by (1) d(z)
```

```
DifferentialLaurentSeriesRing(C)
```

The differential Laurent series ring (in one variable) over the constant field C. If this field is called F, say, then the derivation on F is given by $F.1 \cdot d/d(F.1)$, where $F.1$ is the variable of F.

Example H111E3

This example illustrates the creation and printing of the differential Laurent series ring obtained from the command `DifferentialLaurentSeriesRing`.

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> S;
Differential Ring of Laurent series field in t over Rational Field
with derivation given by Mapping from: Laurent series field in t over Rational
Field to Laurent series field in t over Rational Field given by a rule [no
inverse]
```

```
RingOfFractions(R)
```

Returns the differential ring $R[r^{-1} : r \in R$ not a zero divisor] of fractions of the differential ring R, together with the inclusion map from R to the newly created ring.

```
FieldOfFractions(R)
```

Returns the differential field of fractions of the differential ring R, together with the inclusion map from R to the newly created field.

```
AssignNames(~R, S)
```

Given a differential ring R with n indeterminates and a sequence S of n strings, assign the elements of S to the names of the variables of R.

This procedure only changes the names used in the printing of the elements of R.

111.2.2 Creation of Differential Ring Elements

The easiest way to create an element in a given ring is to use the angle bracket construction to attach names to the indeterminates of the ring. Others are given below.

`Name(R, i)`

`R . i`

The i-th indeterminate of the differential ring R, where i is between 1 and the number of generators of R.

`R ! s`

Coerce the element s in the differential ring R. Elements that are coercible are elements that are coercible in the underlying ring of the differential ring R.

`Zero(R)`

The zero element of the differential ring R.

`One(R)`

`Identity(R)`

The identity element of the differential ring R.

`SeparatingElement(F)`

Returns the separating element of the algebraic differential field F.

Example H111E4

We construct the differential field $F = \mathbf{Q}(z)$ with derivation d/dz and show some of the elements that can be created.

```
> F<z> := RationalDifferentialField(Rationals());
> F.1;
z
> two := F!2;
> two;
2
> Parent(two) eq F;
true
> Zero(F); One(F);
0
1
> Parent(Zero(F)) eq F and Parent(Identity(F)) eq F;
true
> elt := SeparatingElement(F);
> elt;
z
> ISA(Type(elt),RngDiffElt);
true
> Parent(elt) eq F;
```

```
true
> elt eq F!SeparatingElement(UnderlyingRing(F));
true
```

111.3 Structure Operations on Differential Rings

111.3.1 Category and Parent

Differential Rings form the Magma category `RngDiff`. The notional power structures exist as parents of differential rings.

`Category(R)`

`Type(R)`

The category, or type, of the differential ring R.

`Parent(R)`

The power structure of the differential ring R.

111.3.2 Related Structures

The underlying ring and constant ring from which the differential ring was created can each be retrieved as described below. There is also the concept of a base ring. If one has created a differential extension M/F in Magma, then F is the *base ring* of M.

`UnderlyingRing(R)`

The underlying ring of the differential ring R. The type of the underlying ring indicates what ring R inherits from.

`UnderlyingField(R)`

The underlying ring of the differential ring R, provided it is a field.

`BaseRing(R)`

The base ring of the differential ring R.

`BaseField(R)`

The base ring of the differential ring R, provided it is a field.

`ConstantRing(R)`

The constant ring of the differential ring R. The derivation of R acts trivially on the constant ring. It is therefore contained in the differential ring of constants of R.

`ConstantField(R)`

The constant ring of the differential ring R, provided it is a field.

```
ExactConstantField(F)
```

The exact constant field of F, i.e. the algebraic closure in F of the constant field of F, together with the inclusion map to F. The field F must be a function field. The differential field F must have been created with respect to a differential. If the derivation of F has been constructed with respect to a differential, then the exact constant field coincides with the differential field of constants of F.

```
Generators(R)
```

The list of generators of the differential ring R. If there is no list assigned to R, one is constructed by default from the underlying ring of R.

Example H111E5

First we construct the differential field $F = \mathbf{Q}(z)$ with derivation d/dz and show what some of the related structures are. Then we construct the field extension $M = \mathbf{Q}(z, \alpha)$, where α is a root of the polynomial $X^2 - 2$. We do this with the usual `ext< >` constructor. For M we again derive some related structures.

```
> F<z> := RationalDifferentialField(Rationals());
> ConstantRing(F);
Rational Field
> UnderlyingRing(F);
Algebraic function field defined over Rational Field by
$.2 - 4711
>
> _<X> := PolynomialRing(F);
> M<alpha> := ext< F | X^2-2 >;
> BaseRing(M);
Differential Ring of Algebraic function field defined over Rational Field by
$.2 - 4711
with derivation given by (1) d(z)
> BaseRing(M) eq F;
true
> ConstantRing(M);
Rational Field
> E := ExactConstantField(M);
> E;
Number Field with defining polynomial $.1^2 - 2 over the Rational Field
> Generators(M);
[ alpha ]
```

Example H111E6

Related structures also exist for differential Laurent series rings.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> UnderlyingRing(S);
Laurent series field in t over Rational Field
```

```
> ConstantRing(S);
Rational Field
> Generators(S);
[ t ]
```

111.3.3 Derivation and Differential

The derivation of a differential ring and its differential, whenever applicable, can be retrieved as indicated below.

`Derivation(R)`

The derivation of the differential ring R.

`Differential(F)`

The differential belonging to the derivation of the differential field F. The field F must have been constructed in such a way that its derivation is defined by a differential.

Example H111E7

```
> F<z> := RationalDifferentialField(Rationals());
> Derivation(F);
Mapping from: RngDiff: F to RngDiff: F given by a rule [no inverse]
> Differential(F);
(1) d(z)
```

111.3.4 Numerical Invariants

`Ngens(R)`

The number of indeterminates associated with the differential ring R.

111.3.5 Predicates and Booleans

```
R eq F
```

Returns `true` if and only if the differential rings R and F are the same.

```
IsIdentical(R, F)
```

Returns `true` if and only if the differential rings R and F are identical.

```
IsDomain(R)
```

Returns `true` if and only if the differential ring R is a domain.

```
IsField(R)
```

Returns `true` if and only if the differential ring R is field.

```
IsDifferentialField(R)
```

Returns `true` if and only if the ring R is a differential field.

```
IsAlgebraicDifferentialField(R)
```

Returns `true` if and only if the field structure of the differential ring R is an algebraic function field.

```
IsDifferentialSeriesRing(R)
```

Returns `true` if and only if the underlying ring of the differential ring R is a series ring.

```
IsDifferentialLaurentSeriesRing(R)
```

Returns `true` if and only if the underlying ring of the differential ring R is a Laurent series ring and R has been created with a known constant ring.

Example H111E8

This example shows some booleans for various differential rings.

```
> F<z>:=RationalDifferentialField(Rationals());
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> IsAlgebraicDifferentialField(F);
true
> IsDifferentialSeriesRing(F);
false
> IsAlgebraicDifferentialField(S);
false
> IsDifferentialSeriesRing(S);
true
> IsDifferentialLaurentSeriesRing(S);
true
```

`HasProjectiveDerivation(F)`

Returns true if and only if F is a differential ring with derivation weakly of the form $(F.1) \cdot d/d(F.1)$.

`HasZeroDerivation(F)`

Returns true if and only if the algebraic differential field or differential series ring F has zero derivation. When F is a series ring we relax being zero to being weakly zero.

Example H111E9

```
> F<z>:=RationalDifferentialField(Rationals());
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> HasProjectiveDerivation(F);
false
> HasProjectiveDerivation(ChangeDerivation(F,z));
true
> HasZeroDerivation(F);
false
> HasProjectiveDerivation(S);
true
> HasProjectiveDerivation(ChangeDerivation(S,S!3));
false
> HasZeroDerivation(S);
false
```

111.3.6 Precision

`RelativePrecision(F)`

Returns the relative precision of the underlying series ring of F.

`RelativePrecisionOfDerivation(F)`

Given a differential Laurent series ring F, returns the relative precision of the ring derivative of $F.1$.

Example H111E10

This example illustrate the relative precision of differential rings.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> Derivative(t);
t
> IsDifferentialLaurentSeriesRing(S);
true
> RelativePrecision(S);
20
> RelativePrecision(UnderlyingRing(S));
20;
> V<w>:=DifferentialLaurentSeriesRing(Rationals():Precision:=30);
> RelativePrecision(V);
30
> RelativePrecision(V) eq RelativePrecision(UnderlyingRing(V));
true
```

Example H111E11

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> RelativePrecisionOfDerivation(S);
Infinity
> V<w> := ChangeDerivation(S,t+O(t^6));
> Derivation(V)(w);
w^2 + O(w^7)
> RelativePrecisionOfDerivation(V);
5
```

```
ChangePrecision(F, p)
```

Returns the differential series ring isomorphic to F with relative precision p. The map returned is the induced map of F to the new field.

Example H111E12

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RelativePrecision(S);
20
> V<w>,mp :=  ChangePrecision(S,10);
> Type(V);
RngDiff
> IsDifferentialLaurentSeriesRing(V);
true
> RelativePrecision(V);
10
```

```
> RelativePrecision(1/(w-1)) eq 10;
true
> mp(t) eq w;
true
> w@@mp eq t;
true
> derivt := Derivation(S)(t);
> derivt;
t
> derivw := Derivation(V)(w);
> derivw;
w
> mp(derivt) eq Derivation(V)(w);
true
```

111.4 Element Operations on Differential Ring Elements

111.4.1 Category and Parent

`Category(s)`

`Type(s)`

The category, or type, of the differential ring element s.

`Parent(s)`

The parent of the differential ring element s.

111.4.2 Arithmetic

All the usual arithmetic operations are possible for differential ring elements.

`s + t`

The sum of the two differential ring elements s and t.

`-s`

The negation of the differential ring element s.

`s - t`

The difference between the differential ring elements s and t.

`s * t`

The product of the differential ring elements s and t.

```
s ^ n
```

Given a differential ring element s and an integer n, return the n-th power of s. If s is invertible, n may be negative.

```
s div t
```

Given the differential ring elements s and t, return the exact division of s by t, if s is divisible by t.

```
s / t
```

Given the differential field elements s and t, return s divided by t.

111.4.3 Predicates and Booleans

```
s eq t
```

Return `true` iff the differential ring elements s and t are exactly the same.

```
IsZero(s)
```

Return `true` iff the differential ring element s is the zero element of its parent.

```
IsOne(s)
```

Return `true` iff the differential ring element s is the unity element of its parent.

```
IsWeaklyEqual(s, t)
```

Return `true` if and only if the differential ring element s is weakly equal to the differential ring element t.

```
IsWeaklyZero(s)
```

Return `true` if and only if the differential ring element s is weakly equal to the zero element of its parent.

```
IsOrderTerm(s)
```

Return `true` if and only if the differential ring element s is purely an order term of a differential series ring.

Example H111E13

This examples shows the booleans for various differential rings.

```
> F<z> := RationalDifferentialField(Rationals());
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> IsOne(F!1);
true
> t eq t+O(t^2);
false
> IsWeaklyEqual(t, t+O(t^2));
true
> IsWeaklyZero(t^(-1));
false
> IsWeaklyZero(O(t));
true
> IsOrderTerm(t+O(t^2));
false
> IsOrderTerm(O(t));
true
```

111.4.4 Coefficients and Terms

`O(s)`

Creates the order term of the differential series s.

`Truncate(s)`

The known part of the differential series s.

`Eltseq(s)`

Returns the coefficients of the differential ring element s.

`Exponents(s)`

Returns the interval from the valuation of s to (including) the degree of s.

Example H111E14

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> K<x>, mp := ext<F|X^2+X+1>;
> seq := Eltseq(x^2);
> seq;
[ -1, -1 ]
> Universe(seq) eq F;
true
```

Example H111E15

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> O(t+t^2);
O(t)
> Parent(O(t)) eq S;
true
> trunc := Truncate(t^(-1)+5*t^2 +O(t^4));
> trunc;
t^-1 + 5*t^2
> Parent(trunc) eq S;
true
> seq := Eltseq(trunc);
> seq;
[ 1, 0, 0, 5 ]
> Universe(seq) eq Rationals();
true
> Exponents(trunc);
[ -1 .. 2 ]
```

111.4.5 Conjugates, Norm and Trace

```
MinimalPolynomial(s)
```

The minimal polynomial of the differential field element *s* over the base field.

Example H111E16

```
> F<z> := RationalDifferentialField(Rationals());
> P<X> := PolynomialRing(F);
> K<x>, mp := ext<F|X^2+X+1>;
> f := MinimalPolynomial(x^2);
> f;
X^2 + X + 1
> Parent(f) eq P;
true
> g := MinimalPolynomial(x+3/2);
> g;
X^2 + -2*X + 7/4
```

111.4.6 Derivatives and Differentials

`Derivative(s)`

The image of s under the derivation of the parent of s. Notice that it can be different to the “usual” derivative, as it relies on the defined derivation.

`Differential(s)`

Returns the differential of s in the algebraic differential field F, as a differential in the differential space of the underlying ring of F.

Example H111E17

```
> F<z> := RationalDifferentialField(Rationals());
> Derivative(z^2 + 7/z);
(2*z^3 - 7)/z^2
> Differential(z);
(1) d(z)
> Differential(1/z+6+5*z);
((5*z^2 - 1)/z^2) d(z)
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> Derivative(5 + 2*t + 3*t^2);
2*t + 6*t^2
```

111.5 Changing Related Structures

Sometimes whilst working with a differential ring R, one might wish to consider the same ring, but with a different derivation or with a larger constant ring. It is a consequence of the creation of a differential ring, that its constant ring may actually be smaller than its differential ring of constants.

To alter the settings defined by the creation of a differential ring or field the following functions are available.

`ChangeDerivation(R, f)`

Returns a differential ring isomorphic to R, but whose derivation is the map $f\cdot$ `Derivation`(R) induced by the isomorphism. The ring element f must be non–zero. The isomorphism of R to the new differential ring is also returned. The new differential ring has the same underlying ring as R.

Example H111E18

```
> F<z> := RationalDifferentialField(Rationals());
> Derivative(z^2);
2*z
> K, toK := ChangeDerivation(F, z);
> K;
Differential Ring of Algebraic function field defined over Rational Field by
$.2 - 4711
with derivation given by (1/z) d(z)
> toK;
Mapping from: RngDiff: F to RngDiff: K given by a rule
> Derivative(toK(z^2));
2*z^2
> UnderlyingRing(F) eq UnderlyingRing(K);
true
```

Notice that the differential of K is $(1/z)d(z)$, so that the derivation of K is $z \cdot d/dz$, as requested.

```
ChangeDifferential(F, df)
```

Returns the algebraic differential field, whose underlying ring is the one of F, but with derivation with respect to the differential df. The map returned is the bijective map from F into the new algebraic differential field.

Example H111E19

```
> F<z> := RationalDifferentialField(Rationals());
> df := Differential(1/z);
> df in DifferentialSpace(UnderlyingRing(F));
true
> M<u>, mp := ChangeDifferential(F,df);
> IsAlgebraicDifferentialField(M);
true
> Domain(mp) eq F and Codomain(mp) eq M;
true
> Differential(M);
(-1/u^2) d(u)
> mp(z);
u
> Derivation(M)(u);
u^2
> Derivation(F)(z);
1
> dg := Differential(z^3+5);
> N<v>, mp := ChangeDifferential(F,dg);
> Differential(M);
```

```
(3*v^2) d(v)
> mp(z);
v
> Derivation(N)(mp(z));
1/3/v^2
```

```
ConstantFieldExtension(F, C)
```

Returns the differential field isomorphic to the differential field F, but whose constant field is the extension C, and the isomorphism from F to the new field. The differential field F must be an algebraic function field.

Example H111E20

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> M := ext< F | X^2-2 >;
> ConstantField(M);
Rational Field
> _<x>:=PolynomialRing(Rationals());
> C := NumberField(x^2-2);
> Mext, toMext := ConstantFieldExtension(M, C);
> ConstantField(Mext);
Number Field with defining polynomial x^2 - 2 over the Rational Field
> toMext;
Mapping from: RngDiff: M to RngDiff: Mext given by a rule
```

Example H111E21

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> P<T> := PolynomialRing(Rationals());
> Cext := ext<Rationals()|T^2+1>;
> Sext<text>, mp := ConstantFieldExtension(S,Cext);
> IsDifferentialLaurentSeriesRing(Sext);
true
> ConstantRing(Sext) eq Cext;
true
> Derivative(text^(-2)+7+2*text^3+O(text^6));
-2*text^-2 + 6*text^3 + O(text^6);
> mp;
Mapping from: RngDiff: S to RngDiff: Sext given by a rule
> mp(t);
text
```

```
Completion(F, p)
```

Precision RngIntElt *Default :* ∞

The completion of the differential field F with respect to the place p. The place p should be an element of the set of places of F. The derivation of the completion is the one naturally induced by the derivation of F. The map returned is the embedding of F into the completion. Upon creation one can set the precision by using `Precision`. If no precision is given, then a default value is taken.

Example H111E22

This example illustrates the creation of the differential Laurent series ring by using the command `Completion`.

```
> F<z> := RationalDifferentialField(Rationals());
> pl := Zeros(z)[1];
> S<t>, mp := Completion(F,pl: Precision := 5);
> IsDifferentialLaurentSeriesRing(S);
true
> mp;
Mapping from: RngDiff: F to RngDiff: S given by a rule
> Domain(mp) eq F, Codomain(mp) eq S;
true true
> Derivation(S)(t);
1
> 1/(1-t);
1 + t + t^2 + t^3 + t^4 + O(t^5)
```

Example H111E23

This example shows that one does not have to restrict to differential fields of genus 0 to use `Completion`.

```
> F<z> := RationalDifferentialField(Rationals());
> P<Y> := PolynomialRing(F);
> K<y> := ext<F|Y^2-z^3+z+1>;
> Genus(UnderlyingRing(K));
1
> pl:=Zeros(K!z)[1];
> Degree(pl);
2
> S<t>, mp := Completion(K,pl);
> IsDifferentialLaurentSeriesRing(S);
true
> C<c> := ConstantRing(S);
> C;
Number Field with defining polynomial $.1^2 + 1 over the Rational Field
> mp(y) + O(t^4);
```

```
c - t - 4*t^3 + O(t^4)
```

111.6 Ring and Field Extensions

The first differential ring and field extensions we consider are the ones induced by a differential operator. Given a differential operator

$$L = a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0, \quad a_n \neq 0$$

in a differential operator ring $F[D]$ with coefficients in a differential field F, we construct a ring or field extension of degree n over F, whose indeterminates play the role of a formal solution of $L(y) = 0$ and its derivatives.

Given a differential field F, it is also possible to construct differential extensions of the form $F[X]/f(X)$, where $f(X)$ is an irreducible polynomial over F.

```
DifferentialRingExtension(L)
```

Constructs a differential ring extension of the base ring of the differential operator L, by adding a formal solution of L and its formal derivatives as indeterminates.

Let P denote the new differential ring, and F the coefficient ring of L. The ring F is a differential field. If n is the degree of L, the underlying ring of P is a multivariate polynomial ring of degree n over F. We thus have $P = F[Y_1, Y_2, \ldots, Y_n]$, with indeterminates $Y_1, Y_2, \ldots, Y_n$. If L is written as $a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0 \in F[D]$, then the derivation of P is induced by the differential operator L as follows: $\delta_P(Y_i) = Y_{i+1}$, for $i < n$ and $a_n \delta_P(Y_n) = -a_{n-1} Y_{n-1} - \cdots - a_2 Y_2 - a_1 Y_1$. With this construction Y_1 mimics a solution of $L(y) = 0$, and all the others are its derivatives.

```
DifferentialFieldExtension(L)
```

Constructs a differential field extension of the base ring of the differential operator L, by adding a formal solution of L and its formal derivatives as indeterminates.

The construction of the new differential field is completely analogous to the differential ring created by `DifferentialRingExtension(L)`. The only difference is that now a differential field $M = F(Y_1, Y_2, \ldots, Y_n)$, with n indeterminates $Y_1, Y_2, \ldots, Y_n$ is created. The action of the derivation of M on $Y_1, Y_2, \ldots, Y_n$ is as described in `DifferentialRingExtension(L)`.

Example H111E24

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := z^2*D^2-z*D+1;
> P<Y1,Y2> := DifferentialRingExtension(L);
> P;
Differential Ring Extension over F
with derivation given by Mapping from: Polynomial ring of rank 2 over F to
Polynomial ring of rank 2 over F given by a rule [no inverse]
> Derivative(Y1);
Y2
> Derivative(Y2);
-1/z^2*Y1 + 1/z*Y2
```

Example H111E25

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := z^2*D^2-1;
> M<Y,DY> := DifferentialFieldExtension(L);
> IsDifferentialField(M);
true
> Derivative(Y);
DY
> Derivative(DY);
-1/z^2*Y
```

```
ext< F | f >
```

The differential field extension $F(\alpha)$ of the differential field F, where α is a root of the irreducible polynomial f over F. The angle bracket notation may be used to assign the root α to an identifier.

Example H111E26

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> M<alpha> := ext< F | X^2-z >;
> M;
Differential Ring Extension over F by $.1^2 - z
with derivation given by (1) d(z)
> alpha^2;
z
```

The differential of M is the differential dz of the differential space of F lifted to the space of differentials of M.

```
ExponentialFieldExtension(F, f)
```

Returns the differential field $F(E)$ as an extension of F, such that the derivation of E is $f \cdot E$. The parent of f must be F.

```
LogarithmicFieldExtension(F, f)
```

Returns the differential field $F(L)$ as an extension of F, such that the derivation of L is $F(L)!f$. The parent of f must be F.

Example H111E27

```
> F<z> := RationalDifferentialField(Rationals());
> K<E> := ExponentialFieldExtension(F, z);
> K;
Differential Ring Extension over F
with derivation given by Mapping from: Multivariate Rational function field of
rank 1 over F to Multivariate Rational function field of rank 1 over F given by
a rule [no inverse]
> Derivative(E);
z*E
> _<L> := LogarithmicFieldExtension(F, 1/z);
> Derivative(L);
1/z
> Parent($1) eq Parent(L);
true
```

```
PurelyRamifiedExtension(f)
```

Creates a purely ramified field extension M of the differential field F with respect to the purely ramified polynomial $f \in F[X]$. By definition, such a polynomial f is of the form $X^n - a \cdot (F.1)$ for some constant element a in F and positive integer n. The returned extension field M is of the same type as F. The allowed differential fields are algebraic differential fields and differential Laurent series rings. When F is a differential Laurent series ring, its derivation is required to be weakly of the form $c * (F.1) * d/d(F.1)$ for some constant c. The relative precision of M is then n times the relative precision of F. The second argument returned is the embedding map of F into M. The inverse map acts on elements for which it is defined. Otherwise it returns 0.

Example H111E28

A purely ramified extension of an algebraic differential field is constructed in this example.

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> Fext<v>, mp := PurelyRamifiedExtension(X^2-5*z);
> IsAlgebraicDifferentialField(Fext);
true
> mp(z) eq 1/5*v^2;
true
> Parent(mp(z)) eq Fext;
true
> Derivation(Fext)(mp(z));
1
> Derivation(Fext)(v);
1/2/z*v
> Derivation(Fext)(v^2) eq Fext!5;
true
> Inverse(mp)(v^2);
5*z;
```

Example H111E29

A differential Laurent series ring with a derivation without an order term is considered in this example.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> _<T>:=PolynomialRing(S);
> pol := T^4-5*t;
> Sext<r>,mp := PurelyRamifiedExtension(pol);
> IsDifferentialLaurentSeriesRing(Sext);
true
> BaseRing(Sext) eq S and ConstantField(Sext) eq ConstantField(S);
true
> RelativePrecision(Sext);
80
> RelativePrecisionOfDerivation(Sext);
Infinity
> Derivation(S)(t);
t
> mp(t);
1/5*r^4
> Derivation(Sext)(mp(t));
1/5*r^4
> mp(Derivation(S)(t));
1/5*r^4
> x := 4+6*t+O(t^6);
> mp(x);
```

```
4 + 6/5*r^4 + O(r^24)
> Derivation(Sext)(mp(x));
6/5*r^4 + O(r^24)
> mp(Derivation(S)(x));
6/5*r^4 + O(r^24)
> Inverse(mp)(r^4-r^8);
5*t - 25*t^2
> Inverse(mp)(r^4+O(r^5));
5*t + O(t^2)
> Derivation(Sext)(r);
1/4*r
```

Example H111E30

The ring in this example has an order term in its derivation. Therefore, taking a derivative of an element x is of influence on the relative precision of the image of x.

```
> F<z> := RationalDifferentialField(Rationals());
> FF<z>:=ChangeDerivation(RationalDifferentialField(Rationals()),z);
> RR<DD>:=DifferentialOperatorRing(FF);
> RS<DS>, mpRRtoRS :=Completion(RR,Zeros(z)[1]);
> S<t>:=BaseRing(RS);
> IsDifferentialLaurentSeriesRing(S);
true
> _<T> := PolynomialRing(S);
> E<r>, mp := PurelyRamifiedExtension(T^3-5*t);
> IsDifferentialLaurentSeriesRing(E);
true
> RelativePrecision(E);
60
> RelativePrecisionOfDerivation(E);
60
> Derivation(E)(r);
1/3*r + O(r^61);
> mp(t);
1/5*r^3
> Derivation(S)(t);
t + O(t^21)
> Derivation(E)(mp(t));
1/5*r^3 + O(r^63)
> mp(Derivation(S)(t));
1/5*r^3 + O(r^63)
> x:=t^(-2) +7+t^3 +O(t^15);
> Derivation(S)(x);
-2*t^-2 + 3*t^3 + O(t^15)
> Derivation(E)(mp(x));
-50*r^-6 + 3/125*r^9 + O(r^45)
> mp(Derivation(S)(x));
```

```
-50*r^-6 + 3/125*r^9 + O(r^45)
> y := 2*t+O(t^25);
> Derivation(S)(y);
2*t + O(t^21)
> Derivation(E)(mp(y)) eq mp(Derivation(S)(y));
true
> Derivation(E)(mp(y));
2/5*r^3 + O(r^63)
```

111.7 Ideals and Quotient Rings

A differential ideal $I \subset R$ of a differential ring R is an ideal of R that is closed under the derivation of R. However, we consider a differential ideal as an ideal of the underlying ring of R. More specifically, ideals of differential rings are restricted to those rings whose underlying rings are multivariate polynomial rings.

111.7.1 Defining Ideals and Quotient Rings

`DifferentialIdeal(L)`

Given a sequence L with entries in a differential ring R, return the differential ideal generated by the entries of L as an ideal of the underlying ring of R. The underlying ring of R must be of type `RngMPol`. At first the elements of L may generate an ideal which is not closed under the derivation of R. By adding as many derivatives of the elements to the set of generators of the ideal as needed, one obtains a full set of generators for the calculated differential ideal.

`QuotientRing(R, I)`

Given a differential ring R and a differential ideal I, return the differential quotient ring $Q = R/I$. The derivation of Q is induced by the derivation of R. It maps $Q.i$ to $Q!\delta_R(R.i)$, for $i = 1, 2, \ldots, m$ where m is the number of generators of Q (or R). The induced quotient map from R to Q is also returned.

Example H111E31

```
> P := PolynomialRing(Rationals(),1);
> f := map<P->P | a:->a*Derivative(a,1)>;
> R<T> := DifferentialRing(P, f, Rationals());
> L := [T^2+T-1];
> I := DifferentialIdeal(L);
> I;
Ideal of Polynomial ring of rank 1 over Rational Field
Lexicographical Order
Variables: T
Basis:
```

```
[
    T^2 + T - 1,
]
> Q<X>, toQ := QuotientRing(R,I);
> Q;
Differential Ring of Affine Algebra of rank 1 over Rational Field
Lexicographical Order
Variables: X
Quotient relations:
[
    X^2 + X - 1
]
with derivation given by Mapping from: Affine Algebra of rank 1 over Rational
    Field to Affine Algebra of rank 1 over Rational Field given by a rule [no
inverse]
> toQ(T);
X
> Derivative(T^2);
2*T^3
> Derivative(X^2);
X
```

111.7.2 Boolean Operations on Ideals

`IsDifferentialIdeal(R, I)`

Returns `true` if and only if I is a differential ideal of the differential ring R.

111.8 Wronskian Matrix

Let R be a differential ring and let $y_1, y_2, \ldots, y_n$ be elements of R. The *wronskian matrix* of $y_1, y_2, \ldots, y_n$ is defined as the $n \times n$ matrix

$$W(y_1, y_2, \ldots, y_n) = \begin{pmatrix} y_1 & y_2 & \cdots & y_n \\ \delta_R(y_1) & \delta_R(y_2) & \cdots & \delta_R(y_n) \\ \vdots & \vdots & \ddots & \vdots \\ \delta_R^{n-1}(y_1) & \delta_R^{n-1}(y_2) & \cdots & \delta_R^{n-1}(y_n) \end{pmatrix}$$

The *wronskian determinant*, or simply the *wronskian*, of $y_1, y_2, \ldots, y_n$ is the determinant of the wronskian matrix $W(y_1, y_2, \ldots, y_n)$.

`WronskianMatrix(L)`

Given a sequence of differential ring elements L, return the Wronskian matrix of L whose entries are elements of the universe of L.

`WronskianDeterminant(L)`

Given a sequence of differential ring elements L, return the determinant of the Wronskian matrix of L as well as the matrix itself.

Example H111E32

```
> F<z> := RationalDifferentialField(Rationals());
> WronskianMatrix([1,z,z^2]);
[1 z z^2]
[0 1 2*z]
[0 0 2]
> WronskianDeterminant([1,z^2,1/z]);
6/z
[z z^2 1/z]
[1 2*z -1/z^2]
[0 2 2/z^3]
```

111.9 Differential Operator Rings

111.9.1 Creation

`DifferentialOperatorRing(F)`

Returns the differential operator ring over the differential field F.

Example H111E33

```
> F<z> := RationalDifferentialField(Rationals());
> R := DifferentialOperatorRing(F);
> R;
Differential operator ring over Differential Ring of Algebraic function field
defined over Rational Field by
$.2 - 4711
with derivation given by (1) d(z)
```

`AssignNames(~R, S)`

Given a differential operator ring R with n indeterminates and a sequence S of n strings, assign the elements of S to the names of the variables of R.

This procedure only changes the names used in the printing of the elements of R.

111.9.2 Creation of Differential Operators

The easiest way to create an element in a given ring is to use the angle bracket construction to attach a name to the indeterminate of the differential operator ring. Other constructions are given below.

```
Name(R, i)
```

```
R . i
```

The i-th indeterminate of the differential ring R, where i must be 1.

```
R ! s
```

Coerce the element s into the differential operator ring R. Elements that are coercible into R are elements coercible into its underlying ring, sequences, and differential operators defined over the base ring of the coefficient ring of R.

When the base ring of R is an algebraic differential field, elements of other differential operator rings over algebraic differential fields can be coerced into R so long as the underlying rings of the differential fields are the same.

```
Zero(R)
```

The zero element of the differential operator ring R.

```
One(R)
```

The identity element of the differential operator ring R.

Example H111E34

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> R.1;
D
> R!(1/z);
1/z;
>R![1/2,0,5,z];
z*D^3 + 5*D^2 + 1/2
> S<T> := DifferentialOperatorRing(ChangeDerivation(F,z));
> R!T;
z*D
> S!D;
1/z*T
```

111.10 Structure Operations on Differential Operator Rings

111.10.1 Category and Parent

Differential Operator Rings form the Magma category `RngDiffOp`. The notional power structures exist as parents of differential operator rings.

```
Category(R)
Type(R)
```

The category, or type, of the differential operator ring R.

```
Parent(R)
```

The power structure of the differential operator ring R.

111.10.2 Related Structures

As outlined in the introduction, a differential operator ring R is of the form $F[D]$, for a differential ring F. The ring F is called the *base ring* or *coefficient ring* of R.

```
BaseRing(R)
CoefficientRing(R)
```

The base ring, or coefficient ring, of the differential operator ring R.

```
ConstantRing(R)
```

The constant ring of the differential ring operator R.

111.10.3 Derivation and Differential

By construction the variable D of a differential operator ring $F[D]$ is related to the derivation δ_F. That is why δ_F is also considered to be the derivation of R.

```
Derivation(R)
```

The derivation of the differential operator ring R.

```
Differential(R)
```

The differential belonging to the derivation of the differential operator ring R. The derivation must have been constructed in such a way that it is defined by a differential.

Example H111E35

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> BaseRing(R) eq F;
true
> Derivation(R);
Mapping from: RngDiff: F to RngDiff: F given by a rule [no inverse]
> Differential(R);
(1) d(z)
```

111.10.4 Predicates and Booleans

`R eq F`

Returns `true` if and only if the differential operator rings R and F are the same.

`IsIdentical(R, F)`

Returns `true` if and only if the differential operator rings R and F are identical.

`IsDifferentialOperatorRing(R)`

Returns `true` if and only if the given argument is a differential operator ring.

`HasProjectiveDerivation(R)`

Returns `true` iff R is defined over a ring F with derivation weakly of the form $(F.1) \cdot d/d(F.1)$.

`HasZeroDerivation(R)`

Returns `true` iff the base ring of R is an algebraic differential field or a differential series ring F such that the derivation of R acts as a (weak) zero derivation on $F.1$.

Example H111E36

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> IsDifferentialOperatorRing(F);
false
> IsDifferentialOperatorRing(R);
true
> Derivation(R)(z);
1
> HasProjectiveDerivation(R);
false
> HasProjectiveDerivation(ChangeDerivation(R,z));
true
> HasZeroDerivation(R);
false
```

Example H111E37

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> V<W> := DifferentialOperatorRing(S);
> IsDifferentialOperatorRing(V);
true
> Derivation(V)(t);
t
> HasProjectiveDerivation(V);
true
> HasZeroDerivation(V);
false
> P<Q>, mp := ChangeDerivation(V,3/t);
> IsDifferentialOperatorRing(P);
true
> HasProjectiveDerivation(P);
false
> X<y> := BaseRing(P);
> Q*y;
y*Q + 3
```

111.10.5 Precision

`RelativePrecisionOfDerivation(R)`

The relative precision of the derivation of an operator ring over a Laurent series ring.

Example H111E38

This example illustrates the relative precision of derivations of differential operatorrings.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> RelativePrecisionOfDerivation(RS);
Infinity
> RV<DV> := ChangeDerivation(RS, t^2+O(t^8));
> relprec := RelativePrecisionOfDerivation(RV);
> relprec;
6
> RelativePrecisionOfDerivation(BaseRing(RV)) eq relprec;
true
```

111.11 Element Operations on Differential Operators

111.11.1 Category and Parent

`Category(L)`

`Type(L)`

The category, or type, of the differential operator L.

`Parent(L)`

The parent of the differential operator L.

111.11.2 Arithmetic

All the usual arithmetic operations are possible for differential operators. It follows from the multiplication rule for differential operators that the multiplication of differential operators is non–commutative.

`s + t`

The sum of the two differential operators s and t.

`-s`

The negation of the differential operator s.

`s - t`

The difference between the differential operators s and t.

`s * t`

The product of the differential operators s and t.

`s ^ n`

Given a differential operator s and an integer $n \geq 0$, return the n-th power of s.

Example H111E39

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> (z*D-1)*(D+1);
z*D^2 + (z - 1)*D + -1
> (D+1)*(z*D-1);
z*D^2 + z*D + -1
> (D-1/z)^2;
D^2 + -2/z*D + 2/z^2
```

111.11.3 Predicates and Booleans

```
s eq t
```

Return `true` iff the differential operators s and t are exactly the same.

```
IsZero(L)
```

Return `true` iff the differential operator L is the zero element of its parent.

```
IsOne(L)
```

Return `true` iff the differential operator L is the unity element of its parent.

```
IsMonic(L)
```

Return `true` iff the differential operator L is monic.

```
IsWeaklyEqual(L, P)
```

Returns `true` if and only if the differential operator L is weakly equal to the operator P. This means that the i-th coefficients of L and P should be weakly equal to each other for every $i \in [0..\max(\deg(L), \deg(P))]$.

```
IsWeaklyZero(L)
```

Returns `true` if and only if the differential operator $L \in R$ is weakly equal to $R!0$.

```
IsWeaklyMonic(L)
```

Returns `true` if and only if the leading coefficient of the differential operator L is weakly equal to 1.

111.11.4 Coefficients and Terms

Differential operators look like univariate polynomials with coefficients in a differential ring. Some of the terminology used for polynomial rings is mimicked for differential operators.

```
Eltseq(L)
Coefficients(L)
```

Given an operator L with coefficients in R, this function returns the sequence of elements in R, that are the coefficients of L. The sequence is ordered from the constant coefficient to the coefficient of the highest order term of L.

```
Coefficient(L, i)
```

Given an operator L with coefficients in R, this function returns the coefficient of the monomial of degree i in L, as an element of R.

```
LeadingCoefficient(L)
```

Given an operator L with coefficients in R, this function returns the coefficient of the highest order term of L.

`LeadingTerm(L)`

The leading term of the differential operator L.

`Terms(L)`

Given an operator L with coefficients in R, this function returns the sequence of non–zero coefficients of L as elements of R. The sequence is ordered from the lowest order term to the highest order term in L.

Example H111E40

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^3  + (-4*z + 5)*D + (3*z - 4);
> L;
D^3 + (-4*z + 5)*D + 3*z - 4
> Eltseq(L);
[ 3*z - 4, -4*z + 5, 0, 1 ]
> LeadingTerm(L);
D^3
> Terms(L);
[
    3*z - 4,
    (-4*z + 5)*D,
    D^3
]
```

111.11.5 Order and Degree

`Order(L)`

`Degree(L)`

Returns the order of the differential operator L. In the case that L is identically 0, the order is defined to be -1.

`WeakOrder(L)`

`WeakDegree(L)`

If the differential operator L is defined over a differential series ring, then the exponent of the highest coefficient of L that is not weakly 0 is returned.

Example H111E41

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> L := D^2 + 2*t;
> P := O(t)*D^3 + (1+O(t))*D^2  + 2*t;
> Order(L);
2
> Degree(P);
3
> L eq P;
false
> IsWeaklyEqual(L,P);
true
> WeakOrder(P);
2
```

111.11.6 Related Differential Operators

```
MonicDifferentialOperator(L)
```

Given the differential operator L, this function returns the monic differential operator $1/c \cdot L$, where c is the leading coefficient of L.

```
Adjoint(L)
```

Returns the formal adjoint of the differential operator L. The *formal adjoint* of $L = \sum_{i=0}^{n} a_i D^i$ in the differential operator ring $R = F[D]$ over F, is the differential operator $L^* := \sum_{i=0}^{n} (-1)^i D^i * a_i \in R$. It follows from the definition that the orders of L and L^* are the same and that the leading coefficient of L^* is $(-1)^n a_n$.

```
Translation(L, e)
```

If R is the parent of the differential operator L and e is a suitable ring element, then the operator in R obtained by replacing $R.1$ by $R.1 + e$ in L is returned. The second argument returned is the translation map on R by e.

```
TruncateCoefficients(L)
```

If L is defined over a differential series ring, then returned is the operator whose coefficients are the truncations of the coefficients of L.

Example H111E42

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := z*D^3  + (-4*z + 5)*D + (3*z - 4);
> Order(L);
3
> MonicDifferentialOperator(L);
D^3 + (-4*z + 5)/z*D + (3*z - 4)/z
> Adjoint(L);
-z*D^3 + -3*D^2 + (4*z - 5)*D + 3*z
> trans, mp := Translation(L, 2);
> trans;
z*D^3 + 6*z*D^2 + (8*z + 5)*D + 3*z + 6
```

Example H111E43

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L := (5-O(t))*DS^3+(2*t^-1+t^2+O(t^4))*DS - t^-2+t+O(t^3);
> L;
(5 + O(t))*DS^3 + (2*t^-1 + t^2 + O(t^4))*DS + -t^-2 + t + O(t^3)
> TruncateCoefficients(L);
5*DS^3 + (2*t^-1 + t^2)*DS + -t^-2 + t
> L -TruncateCoefficients(L);
O(t)*DS^3 + O(t^4)*DS + O(t^3)
```

111.11.7 Application of Operators

As pointed out in the introduction a differential operator $L = a_nD^n + a_{n-1}D^{n-1} + \cdots + a_1D + a_0$ in $F[D]$ leads to the differential equation $L(y) = 0$ given by

$$L(y) = a_n\delta_F^n(y) + a_{n-1}\delta_F^{n-1}(y) + \cdots + a_1\delta_F(y) + a_0y$$

This notation is formal, but also defines an action of L on any element $y \in F$. The function `Apply` returns the ring element obtained by this action.

```
Apply(L, f)
```

```
L(f)
```

```
f @ L
```

Given a differential operator L and a ring element f, return the ring element obtained after applying L to f, as an element of the base ring of L. The element f must be coercible into the base ring of L.

Example H111E44

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^2-2/z^2;
> Apply(L, z);
-2/z
> L(z);
-2/z
> Apply(L, z^2);
0
```

111.12 Related Maps

This section is devoted to maps between differential operator rings.

`TranslationMap(R, e)`

Returns a map on the differential operator ring R that replaces $R.1$ by $R.1+e$ when applied to a differential operator for some suitable ring element e.

`LiftMap(m, R)`

Let $m : F \to M$ be a differential map on differential fields and R a differential operator ring over F. Then this routine lifts the given map to a map on the differential operator rings $R \to S$, where the basefield of S is M.

Example H111E45

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> transmap := TranslationMap(R, 2 + z);
> Codomain(transmap) eq R;
> transmap(D);
D + z + 2
> transmap(D^2);
D^2 + (2*z + 4)*D + z^2 + 4*z + 5
> P<T> := PolynomialRing(F);
> M<u>, mp := ext<F|T^2+z>;
> liftmap := LiftMap(mp, R);
> Rprime<Dprime> := Codomain(liftmap);
> IsDifferentialOperatorRing(Rprime);
true
> BaseRing(Rprime) eq M;
true
> liftmap(D);
Dprime
```

```
> liftmap(R!z);
z
> Derivation(Rprime)(liftmap(z));
1
> Derivation(Rprime)(u);
1/2/z*u
```

111.13 Changing Related Structures

It may happen that certain intrinsics only work for differential operator rings whose derivations are of a specific form, or whose constant fields have to be large enough. Some of the functions available for changing settings of the differential rings or fields can be used to change the desired related structure on the operator ring directly. To alter some of the settings of a differential operator ring, the following functions are available.

`ChangeDerivation(R, f)`

Returns a differential operator ring isomorphic to R, but whose derivation is given by $f*$ `Derivation(`R`)`. The ring element f must be non–zero. The isomorphism of R to the new differential ring is also returned. The base ring of the new differential operator ring is isomorphic to the one of R, but it has derivation `ChangeDerivation(BaseRing(`R`))`.

`ChangeDifferential(R, df)`

Returns the differential operator ring with differential *df*, and whose underlying ring of its basefield coincides with the one of R. The map returned is the bijective map of R into the new operator ring. The base ring of the new differential operator ring is isomorphic to the one of R. However, the returned inclusion map and taking derivatives may not be commutative.

Example H111E46

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> df := Differential(z^3+5);
> RM<DM>, mp := ChangeDifferential(R,df);
> Domain(mp) eq R and Codomain(mp) eq RM;
true
> M<u> := BaseRing(RM);
> IsDifferentialOperatorRing(RM) and IsAlgebraicDifferentialField(M);
true
> mp(RM!z);
u
> mp(D);
3*u^2*DM
```

```
> D*z, mp(D*z);
u*D + 1
3*u^3*DM + 1
> DM*u;
u*DM + 1/3/u^2
> Differential(RM);
(3*u^2) d(u)
```

```
ConstantFieldExtension(R, C)
```

Returns the ring of differential operators with base ring isomorphic to that of the differential operator ring R, but whose constant field is C. The derivation is extended over the new constant field. The second argument returned is the map from R to the new operator ring.

```
PurelyRamifiedExtension(R,f)
```

When R is a differential operator ring over a differential ring F, this function returns an operator ring over the purely ramified extension of F, as induced by the polynomial f. The polynomial f is of the form $X^n - a \cdot (F.1)$ for some constant element a in F and positive integer n.

Example H111E47

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> _<T> := PolynomialRing(S);
> Rext<Dext>, mp := PurelyRamifiedExtension(R, T^7-3*t);
> Sext<text> := BaseRing(Rext);
> Domain(mp) eq R and Codomain(mp) eq Rext;
true
> IsDifferentialLaurentSeriesRing(Sext);
true
> BaseRing(Sext) eq S;
true
> RelativePrecision(Sext) eq 7*RelativePrecision(S);
true
> D*t;
t*D + t
> mp(D);
Dext
> mp(R!t) eq Rext!(1/3*text^7);
true
> Dext*text;
text*Dext + 1/7*text
```

`Completion(R, p)`

Precision RngIntElt *Default :* ∞

Returns the operator ring $\tilde{R}$, whose base ring is the completion of the base ring of the operator ring R w.r.t. the place p. The second return value is the natural embedding of R into $\tilde{R}$. The precision of the base ring of $\tilde{R}$ can be set by setting `Precision` upon creation. If no precision is set, a default value for the precision is taken.

`Localization(R, p)`

Returns the operator ring whose differential has valuation -1 at p, with derivation $t \cdot d/dt$, where t is the uniformizing element at the place p. The natural map between the operator rings, and the induced image of p are also returned.

`Localization(L, p)`

Given the differential operator L over an algebraic differential field, returns the localized operator of L at the place p. The embedding map between the parents as well as the induced image of the place are also returned.

`Localization(R)`

Given a differential operator ring R over the differential Laurent series ring $C((t))$, returns the operator ring whose derivation is of the form $t \cdot d/dt$, and the natural map between the operator rings.

`Localization(L)`

Given the differential operator L over a differential series ring, returns the localized operator of L and the embedding map between the parents.

Example H111E48

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> D*t;
t*D + t
> Rprime<Dprime>, mp := ChangeDerivation(R,t^2);
> Fprime<tprime> := BaseRing(Rprime);
> mp;
Mapping from: RngDiffOp: R to RngDiffOp: Rprime given by a rule
> Dprime*tprime;
tprime*Dprime + tprime^3
> P<T> := PolynomialRing(Rationals());
> Cext := ext<Rationals()|T^2+1>;
> Rext<Dext>, mp := ConstantFieldExtension(R,Cext);
> Cext eq ConstantRing(BaseRing(Rext));
true
> mp(D);
```

```
Dext
```

Example H111E49

This examples illustrates how to use `Completion` on operator rings.

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> pl := Zeros(z)[1];
> Rcompl<Dcompl>, mp := Completion(R,pl);
> IsDifferentialOperatorRing(Rcompl);
true
> S<t> := BaseRing(Rcompl);
> IsDifferentialLaurentSeriesRing(S);
true
> mp(D);
Dcompl
> Dcompl*t;
t*Dcompl + 1
```

Example H111E50

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> pl := Zeros(z-1)[1];
> Rloc<Dloc>, mp, place:=  Localization(R,pl);
> Domain(mp) eq R, Codomain(mp) eq Rloc;
true true
> place;
(z - 1)
> Differential(BaseRing(Rloc));
(1/(z - 1)) d(z)
> mp(D);
1/(z - 1)*Dloc
> Dloc*(z-1);
(z - 1)*Dloc + z - 1
> L := D + z;
> Lloc, mp, place := Localization(L,Zeros(z)[1]);
> Lloc;
1/z*$.1 + z
```

111.14 Euclidean Algorithms, GCDs and LCMs

A ring of differential operators shares many properties with a univariate polynomial ring. Two of them are GCD and LCM algorithms. However, a consequence of the non–commutative multiplication of a differential operator ring is that the GCD and LCM algorithms cannot be used directly. For instance, in the euclidean algorithm multiplication of the quotient can be done on the left or the right. Therefore one needs to specify the direction of the multiplication in the GCD and LCM algorithms for differential operator rings.

111.14.1 Euclidean Right and Left Division

`EuclideanRightDivision(N, D)`

Given differential operators N and D, return two differential operators Q and R, such that $N = Q \cdot D + R$, with Degree(R) $<$ Degree(D). An error occurs if D is 0.

`EuclideanLeftDivision(D, N)`

Given differential operators D and N, return two differential operators Q and R, such that $N = D \cdot Q + R$, with Degree(R) $<$ Degree(D). An error occurs if D is 0.

Example H111E51

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L1 := D;
> L2 := (D-3)*(D+z);
> EuclideanRightDivision(L1, L2);
0
D
> Q, R := EuclideanRightDivision(L2, L1);
> Q, R;
D + z - 3
-3*z + 1
> L2 eq Q*L1+R;
true
> EuclideanLeftDivision(L2, L1);
0
D
> S, T := EuclideanLeftDivision(L1, L2);
> S, T;
D + z - 3
-3*z
> L2 eq L1*S+T;
true
```

111.14.2 Greatest Common Right and Left Divisors

`GreatestCommonRightDivisor(A, B)`

`GCRD(A, B)`

Given two differential operators $A, B \in R$, return the unique monic differential operator $G \in R$ that generates the left ideal $RA + RB$.

`ExtendedGreatestCommonRightDivisor(A, B)`

Given two differential operators $A, B \in R$, this function returns three operators $G, U, V \in R$, that satisfy $U \cdot A + V \cdot B = G$. The differential operator G is the unique monic right GCD of A and B.

`GreatestCommonLeftDivisor(A, B)`

`GCLD(A, B)`

Given two differential operators $A, B \in R$, return the unique monic differential operator $G \in R$ that generates the right ideal $AR + BR$.

`ExtendedGreatestCommonLeftDivisor(A, B)`

Given two differential operators $A, B \in R$, this function returns three operators $G, U, V \in R$, that satisfy $A \cdot U + B \cdot V = G$. The differential operator G is the unique monic left GCD of A and B.

Example H111E52

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L1 := D^3+z*D^2+D-z;
> L2 := D^2+(z-3)*D-3*z+1;
> GreatestCommonRightDivisor(L1, L2);
D + z
> GreatestCommonRightDivisor(L1, L2) eq GCRD(L1, L2);
true
> G, U, V :=ExtendedGreatestCommonRightDivisor(L1, L2);
> G, U, V;
D + z
1/8
-1/8*D + -3/8
> G eq U*L1+V*L2;
true
> GreatestCommonLeftDivisor(L1, L2);
1
> GCLD(L2,L2*L1) eq L2;
true
```

111.14.3 Least Common Left Multiples

```
LeastCommonLeftMultiple(L)
```

Let $L = D - r$ be a monic operator of degree 1 in $R = F[D]$. Return the least common left multiple of L and all its conjugates over the base ring of F, with respect to the coercion of this base ring into F.

```
LeastCommonLeftMultiple(A, B)
LCLM(A, B)
```

Given two differential operators $A, B \in R$, return the unique monic differential operator $L \in R$, that generates the left ideal $RA \cap RB$. The order of the least common multiple of A and B is at most Order(A)+ Order(B).

```
ExtendedLeastCommonLeftMultiple(A, B)
```

Given two differential operators $A, B \in R$, return three operators $L, U, V \in R$, that satisfy $L = U \cdot A = V \cdot B$. The differential operator L is the unique monic left LCM of A and B.

```
ExtendedLeastCommonLeftMultiple(S)
```

Given the non–empty sequence of differential operators S, this function returns the unique monic left LCM L of the entries of S, as well as a sequence Q of length $\#S$, satisfying $L = Q[i] \cdot S[i]$ for $i = 1, 2, \ldots, \#S$.

Example H111E53

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> LCLM(D, D-z);
D^2 + (-z^2 - 1)/z*D
> L1 := D^3+z*D^2+D-z;
> L2 := D^2+(z-3)*D-3*z+1;
> LeastCommonLeftMultiple(L1, L2);
D^4 + (z - 3)*D^3 + (-3*z + 2)*D^2 + (-z - 3)*D + 3*z - 1
> L, U, V := ExtendedLeastCommonLeftMultiple(L1, L2);
> L, U, V;
D^4 + (z - 3)*D^3 + (-3*z + 2)*D^2 + (-z - 3)*D + 3*z - 1
D + -3
D^2 + -1
> L eq U*L1;
true
> L eq V*L2;
true
> L, Q := ExtendedLeastCommonLeftMultiple([D,D+1,z*D+1]);
> L;
D^3 + (z^2 - 6)/(z^2 - 2*z)*D^2 + (2*z - 6)/(z^2 - 2*z)*D
> Q[3]*(z*D+1) eq L;
true
```

Example H111E54

```
> F<z> := RationalDifferentialField(Rationals());
> P<T> := PolynomialRing(F);
> M<u> := ext<F|T^2+T+1>;
> RM<DM> := DifferentialOperatorRing(M);
> LeastCommonLeftMultiple(DM-u^2);
DM^2 + DM + 1
> lclm := LeastCommonLeftMultiple(DM-u+1);
DM^2 + 3*DM + 3
> EuclideanRightDivision(lclm, DM-u+1);
DM + u + 2
0
> N<v>, mp := ext<F|T^2-z>;
> RN<DN> := DifferentialOperatorRing(N);
> lclm := LeastCommonLeftMultiple(DN-v);
> lclm;
DN^2 + -1/2/z*DN + -z
> LeastCommonLeftMultiple(DN-v, DN+v) eq lclm;
true
> EuclideanRightDivision(lclm,DN-v);
DN + v - 1/2/z
0
> EuclideanRightDivision(lclm,DN+v);
DN + -v - 1/2/z
0
```

111.15 Related Matrices

The *companion matrix* of a monic linear differential operator

$$D^n + a_{n-1}D^{n-1} + \cdots + a_0 \in F[D]$$

is defined as the $n \times n$ matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 1 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 \\ -a_0 & -a_1 & -a_2 & -a_3 & \ldots & -a_{n-1} \end{pmatrix}$$

`CompanionMatrix(L)`

Returns the companion matrix of the monic differential operator L.

Example H111E55

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^3-z*D^2+2*D+5;
> CompanionMatrix(L);
[0 1 0]
[0 0 1]
[-5 -2 z]
```

111.16 Singular Places and Indicial Polynomials

All functions treated in this section concern differential operator rings defined over a function field of transcendence degree one. They all require that the derivation of such an operator ring $F[D]$ over F is defined with respect to a differential.

Any solution of a differential operator $L \in F[D]$ is also a solution of the operator $c \cdot L$, for any non–zero $c \in F$. For considering solutions we may therefore consider L to be monic of the form

$$L = D^n + a_{n-1}D^{n-1} + \cdots + a_0,$$

with coefficients $a_i \in F$ for $i = 1, 2, \ldots n-1$. Each of the coefficients is a rational function in F. They play an important role in the rational solutions of $L(y) = 0$ in F that come to expression in so-called regular and singular places.

111.16.1 Singular Places

Given a place (q) in the set of places of F, a putative rational solution $y \in F$ of $L(y) = 0$ has a q-adic expansion

$$y = y_\alpha\, q^\alpha + y_{\alpha+1}\, q^{\alpha+1} + \ldots$$

It has a pole at the place (q) if the valuation α is negative. After substituting this solution in $L(y)$ it becomes clear that there is only a finite number of places which can occur as poles of an arbitrary solution of $Ly = 0$. Such a place is either a pole of one of the coefficients a_i or a zero or a pole of the differential ω of $F[D]$. There exists a classification for the poles of solutions of $L(y) = 0$.

Given a place (q) and local parameter t at (q), the differential operator can be rewritten as a differential operator $\tilde{L} \in \tilde{F}[\tilde{D}]$, with $\tilde{F} \cong F$, the valuation of whose differential at (q) is 0. The place (q) is defined to be a *singular place* of L, if one of the coefficients of $\tilde{L}$ has negative valuation at (q). Places that are not singular are called *regular.*

There are two kinds of singular places of a differential operator; the *regular singular* places and the *irregular singular* places. With the notation as above, a singular place (q) of L is regular singular if the valuation of the coefficient of $(\tilde{D})^i$ in $\tilde{L}$ is at most $i - n$ for every $i \in \{0, 1, \ldots, n-1\}$. Otherwise it is irregular singular.

`IsRegularPlace(L, p)`

Returns `true` iff the place p is a regular place of the differential operator L. If p is not a regular place of L `false` is returned. This function only works for operators whose derivation is defined by a differential.

`IsRegularSingularPlace(L, p)`

Returns `true` iff the place p is a regular singular place of the differential operator L. If p is not a regular singular place of L `false` is returned. This function only works for operators whose derivation is defined by a differential.

`IsIrregularSingularPlace(L, p)`

Returns `true` iff the place p is an irregular singular place of the differential operator L. If p is not an irregular singular place of L `false` is returned. This function only works for operators whose derivation is defined by a differential.

`SetsOfSingularPlaces(L)`

Two sets are returned. The first set contains precisely all regular singular places of the differential operator L. The second set consists of all irregular singular places of L. This function only works for operators whose derivation is defined by a differential.

`IsFuchsianOperator(L)`

Returns `true` iff the differential operator L is Fuchsian (i.e. if all singular places of L are regular singular). If L is not Fuchsian, `false` is returned. Secondly, the set of all singular places of L, is returned only if L is a Fuchsian differential operator. This function only works for operators whose derivation is defined by a differential.

`IsRegularSingularOperator(L)`

Returns `true` if and only if the differential operator L is regular singular. The operator may be defined over a differential Laurent series ring, F say. In this case being regular singular means that the operator must be regular singular at $F.1$. Then also there is no second argument returned. In the case that the derivation is defined with respect to a differential, then the values from `IsFuchsianOperator(`L`)` are returned.

Example H111E56

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> H := (z^2-z)*D^2+(3*z-6)*D+1;
> IsRegularPlace(H, Zeros(z)[1]);
false
> IsRegularSingularPlace(H, Zeros(z)[1]);
true
> SetsOfSingularPlaces(H);
```

```
{ (1/z), (z - 1), (z) }
{}
> IsFuchsianOperator(H);
true { (z), (1/z), (z - 1) }
> IsFuchsianOperator(D^2-1/z^3);
false
```

Example H111E57

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> IsRegularSingularOperator(D^2 -t*D+2);
true
> IsRegularSingularOperator(D^2 +3);
true
> IsRegularSingularOperator(D^2 +3 +O(t));
true
> IsRegularSingularOperator(D^2 +3*t^(-1));
false
```

111.16.2 Indicial Polynomials

For the definition of a indicial polynomial at a place, we refer to Section 4.1 in [vdPS03].

```
IndicialPolynomial(L, p)
```

Returns the monic indicial polynomial of the differential operator L at the place p. This function only works for operators whose derivation is defined by a differential and whose base ring has one generator.

Example H111E58

```
> F<z> := RationalDifferentialField(Rationals());
> _<T> := PolynomialRing(Rationals());
> R<D> := DifferentialOperatorRing(F);
> H := (z^2-z)*D^2+(3*z-6)*D+1;
> IndicialPolynomial(H, Zeros(z)[1]);
T^2 + 5*T
> IndicialPolynomial(H, Zeros(z-1)[1]);
T^2 - 4*T
> IndicialPolynomial(H, Zeros(1/z)[1]);
T^2 - 2*T + 1
> Apply(H, (z-1)^4/z^5);
0
```

111.17 Rational Solutions

`RationalSolutions(L)`

Given a differential operator L, a basis of the nullspace of rational solutions of $L(y) = 0$ in F is returned as a sequence of basis elements. This function only works for operators whose derivation is defined by a differential. The algorithm that is used is described in Section 4.1 of [vdPS03].

`HasRationalSolutions(L, g)`

Given a differential operator L with coefficients in F and an element g of F, return `true` if there is an element $y \in F$ satisfying $L(y) = g$. If such a solution exists a particular solution in F and the basis of the nullspace of rational solutions in F are also returned. If there is no solution, only `false` is returned. This function only works for operators whose derivation is defined by a differential.

Example H111E59

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> H := (z^2-z)*D^2+(3*z-6)*D+1;
> RationalSolutions(H);
[ (z^4 - 4*z^3 + 6*z^2 - 4*z + 1)/z^5 ]
> L := D^2-6/z^2;
> RationalSolutions(L);
[ z^3, 1/z^2 ]
> Apply(L, z^3+1/z^2);
0
> HasRationalSolutions(L, 6/z);
true -z [ z^3, 1/z^2 ]
> L(-z);
6/z
```

111.18 Newton Polygons

The Newton polygon of a differential operator L and its Newton polynomials can be used to factorize L. A classical example of the Newton polygon uses the derivation $z \cdot d/dz$, where z is a generator of the basefield of L. This Newton polygon is known as *the* Newton polygon of L. Its definition, as used in Magma, is given in §3 of [vH97b] and is applicable to operators over a Laurent series ring with generator z, as well as to operators over fields for which a set of places exist. For fields in the latter category it is however not necessary to restrict to the derivation $z \cdot d/dz$ based at $z = 0$ (in other words: at the place (z)). More generally, the Newton polygon of L at the place (p) is the Newton polygon at $t = 0$ after rewriting L as a differential operator $\tilde{L}$ in a local parameter t of (p), such that derivation of $\tilde{L}$ is of the form $t \cdot d/dt$.

```
NewtonPolygon(L)
```

Returns the Newton Polygon of the differential operator L over a differential Laurent series ring. This means that for the computation of the Newton polygon L may have had to be rewritten as a differential operator $\tilde{L}$ over a differential Laurent series ring $C((t))$, say, such that $\tilde{L}$ has derivation $t \cdot d/dt$. The second argument returned is the operator $\tilde{L}$.

```
NewtonPolygon(L, p)
```

Returns the Newton polygon of the differential operator L at the place p. The derivation of L must be defined with respect to a differential and the base ring of L should have one generator. For the computation of the Newton polygon another differential operator $\tilde{L}$, say, may have had to be calculated. The differential of the derivation of $\tilde{L}$ has valuation -1 at the place p. The differential operator $\tilde{L}$ is also returned.

```
NewtonPolynomial(F)
```

Returns the Newton polynomial of the face F of a Newton polygon. The Newton polygon must have been created with respect to a differential operator. The Newton polynomial depends on a uniformizing element, therefore, its variable is well–defined up to scalar multiplication by a non–zero element. The definition of the Newton polynomial of a face that is used by Magma, is given in Section 3 of [vH97b].

```
NewtonPolynomials(L)
```

Returns all Newton polynomials of L with respect to the faces of its Newton polygon. The second argument returned is the corresponding slopes.

Example H111E60

```
> K := RationalDifferentialField(Rationals());
> F<z> := ChangeDerivation(K, K.1);
> Differential(F);
(1/z) d(z)
```

```
> R<D> := DifferentialOperatorRing(F);
> L := 10*z*D^2+3*D-1;
> npgon, op := NewtonPolygon(L, Zeros(z)[1]);
> npgon;
Newton Polygon of 10*z*$.1^2 + 3*$.1 - 1 over Algebraic function field
defined over Rational Field by
$.2 - 4711 at (z)
> op;
10*z*D^2 + 3*D + -1
> faces:= Faces(npgon);
> faces;
[ <0, 1, 0>, <-1, 1, -1> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
3*T - 1
> NewtonPolynomial(faces[2]);
10*T + 3
```

Example H111E61

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^2+z*D-3*z^2;
> npgon, op := NewtonPolygon(L, Zeros(1/z)[1]);
> op;
1/z^2*$.1^2 + (-z^2 + 1)/z^2*$.1 + -3*z^2
> Differential(Parent(op));
(-1/z) d(z)
> Valuation($1,Zeros(1/z)[1]);
-1
> faces:= Faces(npgon);
> faces;
[ <-2, 1, -2> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
T^2 - T - 3
```

Example H111E62

This example corresponds to Examples 3.46 and 3.49.2 from [vdPS03].

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> L := t*D^2+D-1;
> npgon, op := NewtonPolygon(L);
> L eq op;
true
```

```
> Faces(npgon);
[ <0, 1, 0>, <-1, 1, -1> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomials(L);
[
    T - 1,
    T + 1
]
[ 0, 1 ]
> L := D^2+(1/t^2+1/t)*D+(1/t^3-2/t^2);
> npgon, op := NewtonPolygon(L);
> L eq op;
true
> NewtonPolynomials(L);
[
    T + 1,
    T + 1
]
[ 1, 2 ]
```

111.19 Symmetric Powers

```
SymmetricPower(L, m)
```

Returns the m-th symmetric power of the differential operator L as an element of the parent of L. The symmetric power is monic where possible. If n denotes the order of L, then the degree of the m-th symmetric power of L is at most $\binom{n+m-1}{n-1}$. The algorithm that is used is based on algorithms given in [BMW97].

Example H111E63

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> SymmetricPower(D^2, 3);
D^4
> SymmetricPower(D^3-1, 2);
D^6 + -7*D^3 + -8
```

111.20 Differential Operators of Algebraic Functions

An algebraic function g in a differential field extension M/F satisfies a linear differential equation $L(y) = 0$ with coefficients in $F \subset M$. If the minimal polynomial of g over F is of degree n, then the order of L is at most n.

```
DifferentialOperator(f)
```

Given the irreducible polynomial $f(X) \in F[X]$, return the monic differential operator over F of minimal degree to which a formal root of f is a solution. The field F must be a differential field. The base ring of the created differential operator is F.

The algorithm used in this function is straightforward. If g is a root of an irreducible polynomial $f(X) \in F[X]$, where F is a differential field, then $f(g) = 0$ induces a unique derivation on g. The field $M = F(g)$ is an algebraic differential field extension of F containing all derivatives of g. If n is the degree of the polynomial f, then M/F is a field extension of degree n. This implies that there must be at least one non–trivial linear relation between $g, \delta_M(g), \ldots, \delta_M^n(g)$. The linear relation between these elements involving the lowest powers of δ_M^i gives exactly the desired monic differential operator after a suitable normalization.

Example H111E64

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> f := X^3-z;
> L := DifferentialOperator(f);
> L;
$.1 + -1/3/z
> M<alpha> := ext<F|f>;
> R<D> := DifferentialOperatorRing(M);
> Apply(R!L,alpha);
0
```

111.21 Factorisation of Operators over Differential Laurent Series Rings

When factoring a non-trivial linear differential operator in

$$P[\delta] := k((t))[\delta],$$

with constant field k, differential Laurent Series ring $k((t))$ in t, and derivation δ, one at least wants to compute two operators L and R in P such that $f = L \cdot R$. It is important to distinguish the left and right hand operators (L and R) as such as multiplication generally is non-commutative. When considering differential operators in the operator ring P, it is common to consider δ to be $t \cdot d/dt$. This specific form has the advantage that the degree of $\delta \cdot t$ in t remains one, since $\delta \cdot t = t \cdot \delta + t$. This specific derivation is called the *projective derivation* and we consider this derivation in the rest of this section. In this sense powers of t are eigenvectors under the application of the derivation.

A possible approach for factoring a linear operator would be to compute all non-constant irreducible right hand factors and then use recursion on the appropriate left hand factors. However, this causes a problem as there may be infinitely many factorisations. For instance $\delta^2 - \delta$ has infinitely many factorisations (parametrisations) $(\delta - c/(t+c))(\delta - t/(t+c))$ for any $c \in P^1(k)$. The approach we take to find right hand factors follows [vH97b], and chooses a canonical representative from a class of right hand factors. The obtained representatives can be used in the factorisation of linear differential operators over a rational function field, see [vH97a].

A non-trivial linear differential operator f in P acts on the solution space V of all differential operators in the differential closure of $k((t))$. This action is $\bar{k}$-linear and surjective on V. The kernel $V(f)$ of this map has dimension equal to the order of the operator. The general solution space V can be split into a direct sum of smaller $\bar{k}$-vector spaces V_e. These are minimal such that $f(V_e) \subset V_e$ for every operator in P. Its kernel $V_e(f)$ consists of all solutions of $f(y) = 0$ in V_e. As a consequence the solution space of the operator then is $V(f) = \oplus_e V_e(f)$. For each of the e with non-trivial solution space of f, the idea now is to find an irreducible right hand factor of f in $k((t))[\delta]$ that annihilates $V_e(f)$.

111.21.1 Slope Valuation of an Operator

The Newton Polynomial of an operator f and its slopes contain useful information regarding its factorisation possibilities. It was proved by Malgrange that an operator over a Laurent series ring is reducible if its Newton polygon has at least two slopes. When on the other hand the Newton polygon has one positive slope, and the accompanying Newton polynomial has two relatively prime factors, then the operator is irreducible as well. It can be shown that an irreducible right hand factor has an irreducible Newton polynomial that divides the Newton polynomial of f. When an appropriate irreducible factor of the polynomial of f is taken, One can start building a right hand factor operator with coefficients of a certain precision that has exactly the irreducible factor as its Newton polynomial. Subsequently one can try to lift the coefficients to a better, possibly predescribed precision.

Various measures for the precision are possible, for instance the absolute precision of the coefficients is common. Another valuation metric is or one related to the slope of the Newton polynomial. It is defined as follows. Assume that $P := k((t))[\delta]$ has projective derivation $t \cdot d/dt$, Let s be a rational with numerator n and denominator d such that $\gcd(n, d) = 1$ and $d > 0$ hold. Then the *slope valuation* of a monomial $ct^i\delta^j$, $c \in k\backslash\{0\}$, with respect to s is defined as $V_s(ct^i\delta^j) := id - jn$. The *slope valuation* of the operator $L \in P$ with respect to s subsequently is defined as the minimum of the slope valuation of each non-zero monomial in L w.r.t s. Commonly s is a slope of the Newton polynomial of L. If the operator is the zero operator, the slope valuation is defined to be infinite.

```
SlopeValuation(L,s)
```

Returns the valuation of L, with respect to the rational slope s, when the derivation of L is projective.

Example H111E65

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> P<D>:=DifferentialOperatorRing(S);
> L:=t^(-2)*D^3+t^7;
> SlopeValuation(L,0);
-2
> SlopeValuation(L,1/2);
-7
> SlopeValuation(L,5);
-17
> L:=(0+O(t^6))*D;
> SlopeValuation(L,0);
Infinity
> Valuation(0+O(t^6));
6
> SlopeValuation(P!0,3);
Infinity
```

111.21.2 Coprime Index 1 and LCLM Factorisation

Coprime index 1 factorisation and LCLM factorisation are two factorisation methods that use similar factorisation machinery, but may result in different factorisations, see [vH97b]. Given an operator $f \in k((t))[\delta]$, both compute a right hand factor with respect to each distinct irreducible factor of the Newton polynomial of f. A local lifting procedure with respect to the slope valuation metric is performed to obtain the coefficients of the right hand factors up to a certain accuracy. In addition, no intermediate differential field extensions of $k((t))$ are used.

The coprime index 1 algorithm does not factor an operator f that has Newton polynomial of the form p^n, $n \geq 2$, with slope $s > 0$. The sum of the degrees of the obtained right hand factors of f may be less than the degree of f itself. Their least common left divisor M divides f on the right (i.e. $f = N * M$) with a kernel of dimension less or equal to deg(f). The LCLM algorithm, on the other hand, produces a set of right hand factors of a monic operator f whose LCLM is exactly f up to some precision.

```
Factorisation(L)
Factorization(L)
```

`Precision` RngIntElt *Default :* -1

`Algorithm` MonStgElt *Default :* “*Default*”

Returns a sequence M of operator sequences $[A, B]$ such that $L = A \cdot B$ holds, and B does not have a non-trivial coprime index 1 or LCLM factorisation. As the algorithm sometimes cannot conclude if a right hand factor is irreducible, a second sequence entry $N[i]$ states True if the right hand factor $M[i][2]$ is undisputedly irreducible. The optional argument for the precision is the accuracy up to which the lifting procedure would be performed. The default accuracy is the relative precision of the basering of L. The algorithms used can be eiher "LCLM" or "CoprimeIndexOne". The algorithms used are based on various algorithms in [vH97b].

Example H111E66

The operator $t^2 \cdot d/dt^2 - t \cdot d/dt$ with coefficients in Q has infinitely many factorisations, since it can be written as $(t \cdot d/dt - c/(t + c)) \cdot (t \cdot d/dt - t/(t + c))$ for any rational number c. The factorisation code chooses a canonical right hand factor.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L := D^2 -D;
> factsL,blsL:=Factorisation(L:Algorithm:="LCLM");
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
The number of slopes of the Newton polynomial: 1
> (#factsL eq #blsL) and (#factsL eq 1);
true
> blsL;
[ false ]
> factsL[1];
[
    1,
    D^2 + -1*D
]
> factsL,blsL:=Factorization(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> (#factsL eq #blsL) and (#factsL eq 1);
true
> blsL;
[ true ]
> factsL[1];
[
    D,
    D + -1
]
```

Example H111E67

This example corresponds to Example 3.46 in [vdPS03].

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L:=D*(D+1/t);
> L;
D^2 + t^-1*D + -t^-1
> factsL,blsL:=Factorisation(L:Precision:=4);
Performing coprime index 1 LCLM factorisation.
> blsL;
[ true, true ]
> #factsL eq 2;
true
> factsL[1];
[
    D + t^-1 + 1 - t + 3*t^2 + O(t^3),
    D + -1 + t - 3*t^2 + 13*t^3 + O(t^4)
]
> factsL[2];
[
    D,
    D + t^-1
]
> factsL:=Factorisation(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> #factsL eq 2;
true
> blsL;
[ true, true ]
> [v[1]*v[2]:v in factsL];
[
    D^2 + (t^-1 + O(t^19))*D + -t^-1 + O(t^19),
    D^2 + t^-1*D + -t^-1
]
```

Example H111E68

This example corresponds to Example 3.49 in [vdPS03].

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L:=(D+1/t)*(D+1/t^2);
> L;
D^2 + (t^-2 + t^-1)*D + t^-3 - 2*t^-2
> factsL, blsL:=Factorisation(L:Algorithm:="CoprimeIndexOne",Precision:=6);
Performing coprime index 1 factorisation.
> blsL;
[ true, true ]
> #factsL;
2
> factsL[1];
```

```
[
    D + t^-2 + 2 + t - 3*t^2 - 8*t^3 + O(t^4),
    D + t^-1 - 2 - t + 3*t^2 + 8*t^3 - 9*t^4 + O(t^5)
]
> factsL[2];
[
    D + t^-1,
    D + t^-2
]
> [v[1]*v[2] :v in factsL];
[
    D^2 + (t^-2 + t^-1 + O(t^4))*D + t^-3 - 2*t^-2 + O(t^3),
    D^2 + (t^-2 + t^-1)*D + t^-3 - 2*t^-2
]
```

Example H111E69

This example shows that not all operators may be factored by the factorisation routine. Notice that the Newton polynomial of the operator is a square of a polynomial.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L:=(D+2/t)^2;
> np:=NewtonPolygon(L);
> faces:=Faces(np);
> #faces eq 1;
true
> NewtonPolynomial(faces[1]);
$.1^2 + 4*$.1 + 4
> factsL_lclm,blsL_lclm:=Factorisation(L);
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
> factsL_lclm;
[
    [
        1,
        D^2 + 4*t^-1*D + 4*t^-2 - 2*t^-1
    ]
]
> blsL_lclm;
[ false ]
> factsL_c1,blsL_c1:=Factorisation(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> factsL_c1 eq factsL_lclm;
true
> blsL_c1 eq blsL_lclm;
```

```
true
```

Example H111E70

This example shows that one may not retrieve the factorisation as expected.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L := (D+1/(t+1))*D;
> factsL_c1,blsL_c1:=Factorisation(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> (#factsL_c1 eq 1) and (#blsL_c1 eq 1);
true
> factsL_c1[1][2];
D + O(t^20)
> factsL_lclm, blsL_lclm:=Factorisation(L:Algorithm:="LCLM");
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
The number of slopes of the Newton polynomial: 1
> (#factsL_lclm eq 1) and (#blsL_lclm eq 1);
true
> factsL_c1[1][2], blsL_lclm[1];
D + O(t^20)
false
> M:=(D+1/(t-1))*D;
> factsM:=Factorisation(M:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> # factsM eq 1;
> factsM[1][2]+O(t^4);
D + -1 - 1/2*t - 5/12*t^2 - 3/8*t^3 + O(t^4)
```

Example H111E71

One may wish to adjust the default precision to retrieve enough terms in the coefficients in the factors. This example shows the effect on a rational slope 1/5 on the number of terms in the series coefficients of the factors obtained.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals():Precision:=15);
> R<D>:=DifferentialOperatorRing(S);
> L:=(D^5+t^(-1))*D;
> np:=NewtonPolygon(L);
> Slopes(np);
[ 0 , 1/5 ]
> factsL,blsL:=Factorisation(L:Algorithm:="LCLM");
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
```

```
> bIsL;
[ true, true ]
> [v[2]:v in factsL];
[
    D,
    D^5 + (-1 - t - 63*t^2 + O(t^3))*D^4 + (1 + 3*t + 253*t^2 +
    O(t^3))*D^3 + (-1 - 7*t - 825*t^2 + O(t^3))*D^2 + (1 + 15*t + 2545*t^2
    + O(t^3))*D + t^-1 - 1 - 31*t - 7713*t^2 + O(t^3)
]
> [v[1]*v[2]:v in factsL];
[
    D^6 + t^-1*D,
    D^6 + O(t^3)*D^5 + O(t^3)*D^4 + O(t^3)*D^3 + O(t^3)*D^2 + (t^-1 +
    O(t^3))*D + O(t^2)
]
> factsL:=Factorisation(L:Algorithm:="LCLM",Precision:=75);
Performing coprime index 1 LCLM factorisation.
> [v[1]*v[2]:v in factsL];
[
    D^6 + t^-1*D,
    D^6 + O(t^15)*D^5 + O(t^15)*D^4 + O(t^15)*D^3 + O(t^15)*D^2 + (t^-1 +
    O(t^15))*D + O(t^14)
]
```

111.21.3 Right Hand Factors of Operators

While resorting to field extensions can result in more complex and time consuming computations, they can be used for obtaining irreducible right hand factors over the original base field.

An effective algorithm to obtain a monic irreducible right hand factor of degree one $\tilde{L}$ of $L \in k((t))[\delta]$ in some field extension $\tilde{k}((\tilde{t}))[\tilde{\delta}]$ is presented in [vH97b, §5.1]. Such a factor $\tilde{L} = \tilde{\delta} - r(\tilde{t})$ of L is called a *Ricatti factor* of L. The operator $\mathrm{LCLM}(\tilde{\delta} - \sigma_1(r), \tilde{\delta} - \sigma_2(r), \ldots, \tilde{\delta} - \sigma_m(r))$ where the σ_i are the Galois group elements of $\tilde{k}((\tilde{t}))/k((t))$, then is a monic and irreducible operator invariant under the Galois action. Hence, it naturally reduces to a monic irreducible right hand factor of L.

Other right hand factors of L, that may be defined over a finite field extensions of $k((t))$ are the so-called *semi-regular* parts of L. Such an operator $R_e(L)$ is the monic right hand semi-regular factor of the translation $S_e(L)$ of L by e. Its degree is equal to the dimension of a non-trivial $\bar{k}$-linear vector space $V_e(L)$, and acts as zero on it. In other words $S_{-e}(R_e)$ is a monic right hand factor of L, possibly defined over a field extension of $k((t))$. It can be shown that L is the least common left multiple of all such right hand factors.

The routine `RightHandFactors` returns right hand factors of a given operator possibly by using temporary field extensions. Each of these are canonical representatives of all right hand factors belonging the slopes of the Newton polynomial of the operator. Currently one of the internal routines may fail when performing calculations for some specific right hand factor. In this case we cannot conclude it to be irreducible.

```
RightHandFactors(L)
```

Precision RngIntElt *Default :* -1

The canonical list of monic right hand factors of L, one per slope of the Newton polynomial of L. The i^{th} entry in the second sequence returned is true if the i^{th} right hand factor is undisputedly irreducible. The precision attribute relates to the absolute precision a coefficient in a right hand factor should minimally have.

Example H111E72

This example is Example 3.49 from [vdPS03]. The same right hand factors as in Example H111E68 are obtained.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^2+(1/t^2+1/t)*DS +(1/t^3-2/t^2);
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
> #rhf eq 2;
true
> bl;
[ true, true ]
> (Parent(rhf[1]) eq RS) and (Parent(rhf[2]) eq RS);
true
> lhf,rem := EuclideanRightDivision(L, rhf[1]);
> rem;
O(t^20)
> lhf*rhf[1];
DS^2 + (t^-2 + t^-1 + O(t^22))*DS + t^-3 - 2*t^-2 + O(t^20)
> EuclideanRightDivision(L, rhf[2]);
DS + t^-1
0
```

Example H111E73

This example corresponds to Example 3.52 in [vdPS03]. The answer in the book is erroneous.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^2-3/2*DS+(2*t-1)/(4*t);
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
> bl;
[ true ]
> #rhf eq 1;
true
```

```
> rhf[1] eq L;
true
```

Example H111E74

The operator in this example is the same as in Example H111E69 where the routine `Factorisation` was used. The operator did not factor there, but does when using `RightHandFactors`.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=(DS+2/t)^2;
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Computing a first order Ricatti factor.
Performing LCLM calculation on the Ricatti factor.
> rhf;
[
    DS + 2*t^-1
]
> bl;
[ true ]
```

Example H111E75

This example corresponds to Example 3.53 in [vdPS03].

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^2 +(4+2*t-t^2-3*t^3)/(t^2)*DS+ (4+4*t-5*t^2-8*t^3-3*t^4+2*t^6)/(t^4);
> np:=NewtonPolygon(L);
> faces:=Faces(np);
> #faces eq 1;
true
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
T^2 + 4*T + 4
> factsL, blsL := Factorisation(L);
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
> blsL;
[ false ]
> (#factsL eq 1) and (factsL[1][2] eq L);
true
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
```

```
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Computing a first order Ricatti factor.
Performing LCLM calculation on the Ricatti factor.
> bl;
true
> Degree(rhf[1]);
1
> bl;
true
> Parent (rhf[1]) eq RS;
true
> L - EuclideanRightDivision(L, rhf[1])*rhf[1];
O(t^22)*DS + O(t^20)
```

Example H111E76

A collection of operators having the same Newton polynomial $(T^2+1)(T-1)(T+1)$.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L := DS^4-1/t^4;
> faces:=Faces(NewtonPolygon(L));
> faces;
[ <-1, 1, -4> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
T^4 - 1
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
> bl;
[ true, true, true ]
> [Degree(v) : v in rhf];
[ 1, 1, 2 ]
>  L - EuclideanRightDivision(L, rhf[1])*rhf[1];
O(t^23)*DS^3 + O(t^22)*DS^2 + O(t^21)*DS + O(t^20)
>  L - EuclideanRightDivision(L, rhf[2])*rhf[2];
O(t^23)*DS^3 + O(t^22)*DS^2 + O(t^21)*DS + O(t^20)
>  L - EuclideanRightDivision(L, rhf[3])*rhf[3];
O(t^23)*DS^3 + O(t^22)*DS^2 + O(t^21)*DS + O(t^20)
> M := DS^4-1;
> faces:=Faces(NewtonPolygon(M));
> faces;
[ <0, 1, 0> ]
> NewtonPolynomial(faces[1]);
T^4 - 1
```

```
> rhf, bl := RightHandFactors(M);
Performing coprime index 1 LCLM factorisation.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing LCLM calculation on a semi-regular part.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Computing a first order Ricatti factor.
Performing LCLM calculation on the Ricatti factor.
> rhf;
[
    DS^2 + 1,
    DS + -1
]
> bl;
[ true, true ]
```

Example H111E77

This is the main example of [vH97b]

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^9 + 2*t^-1*DS^8 + 3*t^-2*DS^7 + 2*t^-3*DS^6 + (t^-4 + 2*t^-2)*DS^5 +
> (-3*t^-5 + 5*t^-4)*DS^3 + 3*t^-5*DS^2 + (2*t^-6 + 2*t^-5)*DS + 7*t^-5;
> facts := Factorisation(L);
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
> [Degree(v[2]): v in facts];
[ 1 2 2 4 ]
> isweaklyzero := [];
> vals :=[];
> for i in [1..4] do
>   _,rem := EuclideanRightDivision(L, facts[i][2]);
>   isweaklyzero[i] := IsWeaklyZero(rem);
>   vals[i] := [Valuation(v) : v in Eltseq(rem)];
> end for;
> isweaklyzero;
[ true, true, true, true ]
> [Minimum(v) : v in vals];
[ 14 , 4 , 4 , 11]
> rhf, bl := RightHandFactors(L:Precision:=30);
Performing coprime index 1 LCLM factorisation.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Performing LCLM calculation on a semi-regular part.
```

```
Computation of the LCLM failed.
> bl;
[ true, true, true, false ]
> [Degree(v): v in rhf];
[ 1 2 2 4 ]
> isweaklyzero := [];
> vals := [];
> for i in [1..4] do
>   [Degree(v): v in Eltseq(rhf[i])];
>   _,rem := EuclideanRightDivision(L, rhf[i]);
>   isweaklyzero[i] := IsWeaklyZero(rem);
>   vals[i] := [Valuation(v) : v in Eltseq(rem)];
> end for;
[ 35, 0 ]
[ 35, 35, 0 ]
[ 35, 35, 0 ]
[ 31, 32, 33, 34, 0 ]
> isweaklyzero;
[ true, true, true, true ]
> [ Minimum(v) : v in vals];
[ 30, 30, 30, 27 ]
```

111.22 Bibliography

[**BMW97**] Manuel Bronstein, Thom Mulders, and Jacques-Arthur Weil. On symmetric powers of differential operators. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (Kihei, HI)*, pages 156–163 (electronic), New York, 1997. ACM.

[**vdPS03**] Marius van der Put and Michael F. Singer. *Galois theory of linear differential equations*, volume 328 of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*. Springer-Verlag, Berlin, 2003.

[**vH97a**] Mark van Hoeij. Factorization of differential operators with rational functions coefficients. *J. Symbolic Comput.*, 24(5):537–561, 1997.

[**vH97b**] Mark van Hoeij. Formal solutions and factorization of differential operators with power series coefficients. *J. Symbolic Comput.*, 24(1):1–30, 1997.

PART XV
ALGEBRAIC GEOMETRY

112 SCHEMES

Chapter 112
SCHEMES

112.1 Introduction and First Examples

Schemes are rather general objects of algebraic geometry. A standard reference is Hartshorne's introductory text [Har77]. Included among all schemes are many familiar geometric objects such as plane curves. In Magma, one can work with many of these familiar objects but not with entirely general schemes. Roughly speaking, a scheme in Magma is any geometric object defined by the vanishing of polynomial equations in affine or projective space. In particular, there is no facility for defining a scheme *a priori* in terms of a collection of affine patches. Schemes are not automatically normalized. Maps between schemes can be defined by polynomials or quotients of polynomials.

The sections in this introduction contain examples covering the basic idioms understood by Magma, especially those for creation of geometric objects, and are intended to be the first place of reference for newcomers to this module.

The design of the general scheme module is not particularly subtle or difficult but the philosophy behind it does require a small amount of understanding. There are two things in particular. Firstly, while constructing geometric objects is easy, many of the constructors take an ambient scheme as an argument as well as some polynomials. Thus one's initial step is often to create some large ambient space which is not of primary interest but in which many schemes will lie. In doing so, one usually assigns names to the coordinate functions of this space, and it is these names which are used when writing the polynomials which define some scheme. Secondly, points are not considered to be elements of schemes, but rather elements of one of a series of *point sets* of schemes indexed by the rings containing the coefficients. (Mathematically speaking, these rings are really algebras over the base ring of the scheme, but in Magma the algebra structure is usually implicitly determined by coercion.)

The objects that can be created include

- ambient spaces: affine space, projective space, rational scrolls or weighted projective space over a ring
- schemes as subschemes: the zero locus of polynomials defined on a particular ambient space or on any other scheme
- points of schemes: sequences of ring elements, possibly defined over some extension of the base ring
- maps: sequences of polynomials or rational functions defined on the domain
- linear systems: linear spaces of polynomials defined on ambient spaces

Schemes may be defined quite generally over any ring k, although of course many functions require k to lie in some restricted class. The following restrictions hold: Gröbner basis calculations may only be performed over an exact field or Euclidean domain; resultant calculations may only be carried out over a unique factorization domain; GCD calculations work over any exact field and the integers and over polynomial rings over either of these; the factorisation of polynomials is possible only over the integers, rationals, finite fields, algebraically closed fields, number fields and function fields. Linear systems are based on the linear algebra module in Magma and so are restricted to ambient spaces defined over fields.

The functions described in this chapter are general ones that apply to all schemes. In particular situations additional functions are provided. For example, see Chapter 114 for many specialised functions that apply in the case of curves. In the final subsection of this introduction we say a few words about the various different types of schemes that Magma admits.

To some extent we try to emulate Hartshorne's text [Har77] although we remain only a fraction of the way through the material of that book and, of course, we have made innumerable compromises.

In the examples below, `>` at the start of a line is the Magma prompt. It is followed by input which may be typed into a Magma session. The remaining lines are output which has been edited slightly in some circumstances, but should nonetheless match closely what appears on screen.

112.1.1 Ambient Spaces

Most schemes are considered to live in an *ambient space.* These include affine and projective spaces, rational scrolls and products. They may be defined over any base ring. The main technical point is that they have an associated coordinate ring (or a homogeneous coordinate ring) that is a polynomial ring, possibly with one or more gradings associated to it.

The syntax for defining an affine space over a ring is similar to that employed when defining a polynomial ring over another ring. The angled bracket notation is used for assigning names to the coordinate functions. It is optional as in the case of polynomial rings. We illustrate by creating an affine 3-space over the finite field of 23 elements with coordinates x, y, z.

Example H112E1

```
> k := FiniteField(23);
> A<x,y,z> := AffineSpace(k,3);
> A;
Affine Space of dimension 3
Variables : x, y, z
```

Various attributes of A are cached and may be subsequently retrieved.

```
> BaseRing(A);
Finite field of size 23
> Dimension(A);
3
```

```
> A.1;
x
> CoordinateRing(A);
Polynomial ring of rank 3 over GF(23)
Lexicographical Order
Variables: x, y, z
```

Projective spaces are normally defined in the same way, but they can also be defined with weights.

```
> P<u,v,w> := ProjectiveSpace(k,[1,2,3]);
> P;
Projective Space of dimension 2
Variables : u, v, w
Gradings :
1       2       3
```

Having defined an ambient space A, polynomials in its coordinate functions can be created. These polynomials are elements of the coordinate ring of A. These polynomials (and sometimes also quotients of them) can be used to define geometric objects related to A. Elements of other polynomial rings have no meaning on A and their use will result in an error.

112.1.2 Schemes

Subschemes of ambient spaces prescribed by the vanishing of finitely many polynomials may be defined. Just as a polynomial ideal in Magma belongs to some polynomial ring, so any scheme defined by polynomials is contained in some ambient space. In the case of ideals of polynomial rings the function `Generic()` recovers the polynomial overring. For schemes the analogous function is `AmbientSpace()` (or `Ambient`) which recovers the ambient space.

At the time a scheme is created the system only checks that the defining equations make sense — that they are defined on the nominated ambient space, and are homogeneous if necessary — and does not check other properties such as whether or not the scheme is empty.

In this example, the twisted cubic curve in projective 3-space is defined in terms of equations. The constructor takes these equations in a sequence as one of its arguments.

Example H112E2

```
> k := Rationals();
> P<u,v,w,t> := ProjectiveSpace(k,3);
> M := Matrix(CoordinateRing(P),2,3,[u,v,w,v,w,t]);
> eqns := Minors(M,2);
> C := Scheme(P,eqns);
> C;
Scheme over Rational Field defined by
u*w - v^2
```

```
-u*t + v*w
v*t - w^2
> AmbientSpace(C);
Projective Space of dimension 3
Variables : u, v, w, t
> Dimension(C);
1
> IsNonsingular(C);
true
```

In fact, it is possible to create schemes without reference to an ambient space. For instance, the intrinsic `Spec` may be applied to an affine algebra. But even so, the ambient space defined by the polynomial overring of that algebra is created in the background and may be recovered using the intrinsic `Ambient`. Many constructors require a reference to some overscheme to be clear about *which* scheme the new object is meant to live in.

We note here that there is an important difference between affine schemes and projective ones.

For affine schemes, the defining ideal (generated by the defining equations) is *unique.* That is, there is a $1 - 1$ correspondence between subschemes of an affine ambient space and the ideals of the coordinate ring of that ambient.

On the other hand, for projective ambients, a given subscheme is defined by multiple homogeneous ideals of the ambient coordinate ring. In this case however, there is a *largest* defining ideal for each subscheme, which we refer to as the *saturated* one.

The practical effect of this is that Magma may have to replace the original defining ideal of a projective scheme with the saturated ideal to guarantee the correct result of certain functions (see section 112.3 on page 3490 for more details).

112.1.3 Rational Points

Although closed points of schemes may be defined as schemes in terms of polynomials, there is a far more convenient way to define them: simply coerce the coordinates of the point into the scheme. This is to allow points to be used in a mathematically colloquial way: one understands the statement "the point p lies on the curve C" to mean that the coordinates of p satisfy the equation of C, rather than to mean an inclusion between the two defining ideals. (It also avoids the ideal inclusion test which would be a much more expensive calculation.)

The points $p = (-1, 1)$, $q = (1, 2)$ are created in the affine plane over the finite field of 31 elements. Magma's coercion operator, the `!` symbol, provides a concise notation for specifying natural reinterpretations of objects. In this case a sequence of integers is reinterpreted as a point of the finite plane. One of the points created below lies on the standard parabola C.

Example H112E3

```
> k := GF(31);
> A<x,y> := AffineSpace(k,2);
> p := A ! [-1,1];
> q := A ! [1,2];
> p,q;
(30, 1) (1, 2)
> C := Scheme(A,y-x^2);
> p in C;
true (30,1)
> q in C;
false
> [-1,1] in C;
true (30, 1)
```

But this is only the beginning of the story. Objects in Magma are always considered to lie in some set or structure called their *parent*. Although it would be natural to take the scheme as the parent of points, instead points have a *point set* as their parent. Point sets are the Magma equivalent of "L-valued points" of schemes. If k is the base ring of a scheme X and L is some k-algebra, then the point set of X over L, denoted $X(L)$, comprises points with coordinates in L. In the previous example, the sequences of coordinates were defined over the base ring k and the coercion created elements in the point set $A(k)$. Predicates such as `p in C` were evaluated by testing whether the coordinates of p satisfied the equations of the scheme rather than by consulting their parents. If the point does happen to lie on C, then p is returned as a point of C as second return value. Note the difference between the apparently identical points $(30, 1)$: the first, (p), lies in a point set of A while the second lies in a point set of C. The same effect was achieved using the sequence alone in the last line.

Point sets thus allow one to define points over extensions of k without having to define a new scheme over that extension. In the next fragment, we show how to make a point of C over an extension.

```
> k1<w> := ext< k | 2 >;
> C(k1) ! [w^16,3];
(w^16, 3)
```

An error is signalled if a point set is not nominated.

```
>  C ! [w^16,3];
>>   C ! [w^16,3];
       ^
Runtime error in '!': Illegal coercion
LHS: Sch
RHS: [FldFinElt]
```

The moral is:

A point of a scheme is created by coercing a sequence of coordinates into a point set of the scheme.

While it is true that the "scheme ! coordinates" operation applies when the sequence is defined over the base ring, it might best be thought of as a convenient shorthand in predictable and simple situations. This is analogous to the situation when constructing sets or sequences.

```
> C(k1) ! [w,w^2];
(w, w^2)
>  C(k) ! [w,w^2];
>> C ! [w,w^2];
     ^
Runtime error in '!': Illegal coercion
```

This coercion fails since the coordinates do not belong to a field that embeds into k.

112.1.4 Projective Closure

Affine schemes have projective closures and projective schemes have standard affine patches. For example, the projective plane has three standard affine patches, each of which may be recovered as illustrated in the following example. Here we compute the third affine patch, that is the patch whose points have nonzero first coefficient (in the projective space).

Example H112E4

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> A := AffinePatch(P,3);
> A;
Affine Space of dimension 2
Variables : $.1, $.2
> A<u,v> := A;
> A;
Affine Space of dimension 2
Variables : u, v
> P eq Codomain(ProjectiveClosureMap(A));
true
```

Note that variable names on the patch, and also on closures, are not created automatically. Also, the relationship between patch and closure is cemented by a map.
The projective closures of all schemes contained in a single affine space will lie in a common projective space. Moreover, the closures of schemes lying in distinct affine patches of a single projective space will lie in that same space. In particular, the projective closures of different patches of a projective scheme will be identical.

```
> X := Scheme(A,[u^2-v^3,u^2+v^3]);
> PX := ProjectiveClosure(X);
> PX;
Scheme over Rational Field defined by
z^3
y^2
```

```
> AffinePatch(PX,3) eq X;                     // (1)
true
> aX1:=AffinePatch(PX,1);
> IsEmpty(aX1);                               // (2)
true
> ProjectiveClosure(aX1) eq PX;               // (2)
true
> Y := Scheme(P,[x*y^2-z^3,x*y^2+z^3]);
> AffinePatch(Y,3);
Scheme over Rational Field defined by
u^2 - v^3
u^2 + v^3
> AffinePatch(Y,3) eq X;                      // (3)
true
> Y eq PX;
false
```

This example shows several things. First it shows that taking the appropriate affine patch of a projective closure returns the same scheme again (1). Second, it shows that the projective closure of an affine patch always returns the exactly the same scheme and not a newly created version. This is not always mathematically correct and means that projective closure is dependent on how a scheme was created. Finally, it shows that projective schemes are "saturated at infinity", thereby removing the unnecessary x factor (3).
Points are handled cleanly with respect to projective closure.

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

Although the coercion of points is very flexible — the affine point p can usually be used in place of the projective point `PX ! p` even when working in PX — code which is explicit about this kind of coercion is probably more clear.

112.1.5 Maps

Maps between schemes are defined in terms of either polynomials or rational polynomials. When a function field exists for the domain, function field elements may also be used. When the domain or codomain is projective, that is, has at least one grading, then compatibility of the defining functions with the gradings will be checked.

Maps to projective spaces are normalised by clearing polynomial denominators.

Example H112E5

```
> k := Rationals();
> P1<s,t> := ProjectiveSpace(k,1);
> P2<x,y,z> := ProjectiveSpace(k,2);
> f := map< P1 -> P2 | [s/t,s^2/(s^2 - t^2),t/s] >;
> f;
```

```
Mapping from: Prj: P1 to Prj: P2
with equations :
s^4 - s^2*t^2
s^3*t
s^2*t^2 - t^4
> IsRegular(f);
true
> Image(f);
Scheme over Rational Field defined by
-x^3*z + x^2*y^2 - 2*x*y^2*z + y^2*z^2
```

The images of points may be computed in the natural way:

```
> p := P1 ! [3,2];
> f(p);
(9/4 : 27/10 : 1)
> f(p) in Image(f);
true (9/4 : 27/10 : 1)
```

And schemes may be pulled back by maps.

```
> S := Scheme(P2,x^2 - y*z);
> Z := Pullback(f,S);
> Z;
Scheme over Rational Field defined by
s^8 - 2*s^6*t^2 - s^5*t^3 + s^4*t^4 + s^3*t^5
> RationalPoints(Z);
{@ (1 : 1), (-1 : 1), (0 : 1) @}
> P := PointsOverSplittingField(Z); P;
{@ (0 : 1), (-1 : 1), (1 : 1), (r1 : 1), (r2 : 1), (r3 : 1) @}
> Ring(Universe($1));
Algebraically closed field with 3 variables
Defining relations:
[
    r3^3 - r3 - 1,
    r2^3 - r2 - 1,
    r1^3 - r1 - 1
]
```

This code computes the intersection of the scheme which is image of f with the conic S (although note that S is defined only as a scheme here and not as a conic). Moving to an algebraic closure one sees a Galois 3-cycle of points among the rational points. Additional functions for analysis of the multiplicities of these intersections are described in Chapter 114.

112.1.6 Linear Systems

Linear systems in projective space are simply collections of hypersurfaces having a common degree which are parametrised linearly by a vector space. The set of all conics in the plane is an example, being parametrised by the 6-dimensional vector space of possible coefficients of a general conic:

$$(a, b, c, d, e, f) \longleftrightarrow (ax^2 + bxy + cy^2 + dxz + eyz + fz^2 = 0).$$

A linear system in Magma has associated with it a fixed basis of forms, called its *sections*, of the given degree which allows vectors of coefficients to be interpreted as hypersurfaces. In the example above, the basis is the set of monomials of degree 2 and the bijection between vectors of coefficients and hypersurfaces is explicit. Linear systems are closely related to maps since their sections may be used to define a map from the space on which they are defined to some other projective space.

Let P be a projective plane over some field k with coordinates x, y, z. The linear system of conics on P is created.

Example H112E6

```
> Q := RationalField();
> P<x,y,z> := ProjectiveSpace(Q,2);
> L := LinearSystem(P,2);
> L;
Linear system on Projective Space of dimension 2
Variables : x, y, z
with 6 sections: x^2 x*y x*z y^2 y*z z^2
> Sections(L);
[ x^2, x*y, x*z, y^2, y*z, z^2 ]
```

These sections are now used to make a map from the plane P to 5-space. Its image is the *Veronese surface.*

```
> P5<u0,u1,u2,u3,u4,u5> := ProjectiveSpace(Q,5);
> phi := map< P -> P5 | Sections(L) >;
> Image(phi);
Scheme over Rational Field defined by
-u0*u3 + u1^2
-u0*u4 + u1*u2
-u0*u5 + u2^2
-u1*u4 + u2*u3
-u1*u5 + u2*u4
-u3*u5 + u4^2
```

Geometrical conditions may also be imposed on a linear system. For example, given a linear system L and a point p lying in the projective space on which L is defined, a subsystem of L consisting of those hypersurfaces of L which contain p may be defined implicitly. It may be checked (choosing more convenient coordinates) that the image of this new system is a projection of the Veronese surface from the point $f(3, 2, 1) = (9, 6, 4, 3, 2, 1)$ lying on it. This is an embedding of the blowup of the plane P at the point $(3, 2, 1)$ also known as the *cubic scroll.*

```
> p := P ! [3,2,1];
> L1 := LinearSystem(L,p);
> L1;
Linear system on Projective Space of dimension 2
Variables : x, y, z
with 5 sections:
x^2 - 9*z^2
x*y - 6*z^2
x*z - 3*z^2
y^2 - 4*z^2
y*z - 2*z^2
> P4<v0,v1,v2,v3,v4> := ProjectiveSpace(Q,4);
> phi := map< P -> P4 | Sections(L1) >;
> Image(phi);
Scheme over Rational Field defined by
-v0*v3 + v1^2 - 4*v2^2 + 12*v2*v4 - 9*v4^2
-v0*v4 + v1*v2 - 2*v2^2 + 3*v2*v4
-v1*v4 + v2*v3 - 2*v2*v4 + 3*v4^2
```

It may also be checked that this image can be described as the vanishing of the two by two minors of a certain matrix.

```
> M := Matrix(3,[v0, v1 + (2*v2 - 3*v4), v2, v1 - (2*v2 - 3*v4), v3, v4]);
> Minors(M,2);
[
    v0*v4 - v1*v2 + 2*v2^2 - 3*v2*v4,
    v1*v4 - v2*v3 + 2*v2*v4 - 3*v4^2,
    v0*v3 - v1^2 + 4*v2^2 - 12*v2*v4 + 9*v4^2
]
> ideal< CoordinateRing(P4) | $1 > eq Ideal(Image(phi));
true
```

112.1.7 Aside: Types of Schemes

This section gives a general overview of the types of scheme that Magma admits. While it is not necessary to know this for most applications, it is a useful guide to our point of view and gives some indication of the different data structures used.

The main type is `Sch`. Very general functions such as `BaseRing()` apply at this level. Gröbner basis calculations also apply. One level below this there are

`Aff` — a type for affine spaces

`Prj` — a type for projective spaces

`Crv` — a type for curves

`Clstr` — a type for zero-dimensional schemes, or clusters

These inherit all the operations which apply at the level of `Sch`. The affine type is straightforward. The projective type contains spaces with graded polynomial coordinate rings. There are a number of subtypes of this which identify spaces with more than one grading.

Objects of the curve type `Crv` currently only include schemes defined by a single nontrivial equation in a two-dimensional ambient space. Functions which apply to the curve type are detailed in Chapter 114. The most powerful of those require certain irreducibility and separability conditions to be satisfied by the equation.

More specialised curve types are derived from `Crv`. They include `CrvCon` for plane conics especially those defined over the rationals where fast point-finding algorithms exist; `CrvRat` for rational curves for which a parametrisation algorithm exists; `CrvEll` for elliptic curves, probably the most sophisticated part of the entire geometry module; and `CrvHyp` for hyperelliptic curves where there are fast algorithms for computing on the jacobian. Each of these has a handbook chapter which presents their specialised functions.

112.2 Ambients

For the purposes of this chapter, any scheme is contained in some ambient space, either an affine space or one of a small number of standard projective spaces: these are projective space itself, possibly weighted, and rational scrolls. The basic property of these spaces is that they have some kind of coordinate ring that is a polynomial ring. It happens again and again that we lift polynomials to these polynomial rings before working with them. It is possible to define schemes without reference to such an ambient space, but one will be created in the background in any case.

Listed in this section are the basic creation methods for ambient spaces. Names for the coordinates will usually be required for creating polynomials later on. Names of coordinate functions may be defined using the diamond bracket notation in the same way as for polynomial rings. Coordinate names defined using this will be *globally* defined and retained even outside the context in which they were set. Alternatively, explicit naming functions may be used after creation.

112.2.1 Affine and Projective Spaces

These are the basic ambient spaces. They are used in many situations and are usually sufficient, although there are more in the next section.

```
AffineSpace(k,n)
```

Create an n-dimensional affine space over the ring k. The integer n must be positive. Names can be assigned using the angle bracket notation, e.g. `A<x, y, z> := AffineSpace(Rationals(), 3)`, which will assign the names to the coordinate ring, usually a multivariate polynomial ring, in the same way as the angle bracket notation works for the multivariate polynomial rings.

`ProjectiveSpace(k,n)`

`ProjectiveSpace(k,W)`

Create an n-dimensional projective space over the ring k. The integer n must be positive. The second argument to this intrinsic can be a sequence of positive integer weights. These weights will be assigned to the coordinate functions of the space. The dimension of the space is one less than the length of this sequence. (At present there are very few functions to perform analysis on weighted projective spaces, but maps between them are treated correctly.) Names can be assigned using the angle bracket notation, e.g. `P<x, y, z> := ProjectiveSpace(Rationals(), 2)`, which will assign the names to the coordinate ring, usually a multivariate polynomial ring, in the same way as the angle bracket notation works for multivariate polynomial rings.

`AffineSpace(R)`

`Spec(R)`

Create the affine space whose coordinate ring is the multivariate polynomial ring R. The coordinate names for the affine space will be inherited from R.

`ProjectiveSpace(R)`

`Proj(R)`

Create the projective plane whose homogeneous coordinate ring is the multivariate polynomial ring R. If R has been assigned a grading then that grading will be used otherwise it will be considered to have the standard grading by degree.

`AssignNames(~A,N)`

A procedure to change the print names of the coordinate functions of the ambient space A. It leaves A unchanged except that the visible names of the first $\#N$ coordinate functions are replaced by the strings of N and the rest return to their default. It does *not* assign the coordinate functions themselves to any identifiers. That must be done by hand, for instance by the command `x := A.1;`. Note that this will change the variable names of the coordinate (polynomial) ring.

`A . i`

`Name(A,i)`

The ith coordinate function of A as an element of the coordinate ring of A.

Example H112E7

An affine 3-space over the finite field of 11^2 elements is created. Initially only the first two coordinate functions are named. The third adopts the default name `$.3`. The identifier which refers to it is `A.3`. Then new names are assigned to all coordinate functions and new identifiers set to refer to them. (Check what happens if the identifiers are not assigned.) Notice that the previous identifiers, x, y, are not erased, although their print values have been updated.

```
> A<x,y> := AffineSpace(FiniteField(11,2),3);
> A;
Affine Space of dimension 3
Variables : x, y, $.3
> AssignNames(~A,["u","v","w"]);
> u := A.1; v := A.2; w := A.3;
> A;
Affine Space of dimension 3
Variables : u, v, w
> x;
u
> u eq x;
true
```

Print values are global in Magma, meaning that even if they are changed in the local environment of a function or procedure the new names will persist.

```
A eq B
```

Returns `true` if and only if the ambient spaces A and B are identical. This will only be the case if both A and B refer to the same instance of creation of the space.

112.2.2 Scrolls and Products

These spaces are created using multiple gradings. They are not as fundamental as the affine and projective spaces of the previous section and may be passed over on first reading.

As we have said, the important thing about ambient spaces in this system is that their coordinate rings are essentially polynomial rings. For affine spaces, this is literally true. For projective spaces, one talks about the *homogeneous* coordinate ring and restricts attention to homogeneous polynomials, that is, polynomials whose terms all have the same weight with respect to a single grading, but nonetheless one is working inside a polynomial ring.

This trick can be pushed further by admitting more than one grading on a polynomial ring. The standard example of this is the family of rational ruled surfaces, or rational surface scrolls, which have a bihomogeneous coordinate ring with four variables u, v, x, y and two gradings which are often chosen to be

$$[1, 1, -n, 0] \quad \text{and} \quad [0, 0, 1, 1]$$

for some nonnegative integer n. Now we restrict attention to polynomials of some homogeneous *bidegree* (or more generally, *multidegree*) of the form $[a, b]$. As with ordinary projective space, polynomials of a given bidegree form a vector space with a favourite basis of monomials. For example, if $n = 3$, then monomials of bidegree $[2, 1]$ have as their basis

$$u^5x, u^4vx, \dots, v^5x, uy, vy.$$

As with ordinary projective space, one can ask which polynomials have negative degree. The difference is that this time there is just a little room for manoeuvre: with the same n the monomials of degree $[-4, 2]$ are

$$u^2x^2, uvx^2, v^2x^2.$$

We consider the geometric object having this coordinate ring to be a primitive object of the system. It is called a ruled surface and denoted $\mathbf{F}_n$. This seems to be a rather implicit approach at first, but is very similar to the way one works with projective space itself: at first one learns about homogeneous polynomials, degree 0 rational polynomials, affine patches and so on, but once used to that one gets on and uses it.

There are affine patches on $\mathbf{F}_n$ — a standard 'first' patch is the affine plane where $v \neq 0$ and $y \neq 0$ — and one can study them to get an idea of what this surface looks like. In this case it is a $\mathbf{P}^1$ bundle over $\mathbf{P}^1$, the map to $\mathbf{P}^1$ given by the bidegree $[0, 0]$ ratio u/v and the fibre coordinate being the ratio x/y, not of trivial bidegree on the whole surface, but well-defined for any fixed value of u/v.

We think of the gradings as forming a matrix and allow more general scrolls arising from matrices which are partially row reduced to have rows of the form

$$\begin{pmatrix} 1 & \dots & 1 & * & \dots & * & \dots & & & & & \\ 0 & \dots & 0 & 1 & \dots & 1 & * & \dots & & & & \\ \vdots & & & & & & & & & & & \\ 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & \dots & 1 \end{pmatrix}$$

where the $*$ entries may be any integers. To get products of projective spaces we simply set all the entries to be 0. Thinking about which multihomogeneous polynomials arise in this ring and comparing with functions on products of projective spaces is a first exercise.

A reference for the point of view we take on these scrolls is [Rei97] Chapter 2.

`DirectProduct(A,B)`

The product of the ambient spaces A and B together with a sequence containing the two projection maps.

`RuledSurface(k,a,b)`

This returns the ruled surface defined over the ring k whose negative section has selfintersection $\pm(a - b)$, where a, b are non-negative integers.

```
RuledSurface(k,n)
```

If n is a nonnegative integer, this returns the ruled surface defined over the ring k whose negative section has selfintersection $-n$. The integer n must be non-negative. In terms of the gradings, this means using the standard gradings as described in the introduction with the top-right-hand entries being $-n, 0$.

```
AbsoluteRationalScroll(k,N)
```

If N is a sequence nonnegative integers this returns the rational scroll with base ring k and gradings with entries being $-N$.

```
ProductProjectiveSpace(k,N)
```

If $N = [n_1, \ldots, n_r]$ is a sequence of positive integers this returns the product of ordinary projective spaces

$$\mathbf{P}^{n_1} \times \ldots \times \mathbf{P}^{n_r}$$

of dimensions of N over the ring k. This does not create independent copies of the projective factors and in particular does not return projection maps to the factors.

112.2.3 Functions and Homogeneity on Ambient Spaces

```
CoordinateRing(A)
```

The coordinate ring of the ambient space A. This is some polynomial ring of appropriate rank over the base ring. Gradings on this ring are usually independent of those of the scheme. Note that if the coordinate ring has zero rank then it will be the base ring.

```
FunctionField(A)
```

The function field of the ambient space A. This is a field isomorphic to the field of fractions of the coordinate ring of A.

```
HasFunctionField(A)
```

```
Gradings(X)
```

A sequence containing all the gradings on the projective space X. Each such grading is a sequence of integers whose length is the same as the number of coordinate functions of X. The same sequence is returned when this function is applied to any scheme contained in X.

```
NumberOfGradings(X)
NGrad(X)
```

The number of independent gradings on the projective space X. The same number is returned when this function is applied to any scheme contained in X.

```
NumberOfCoordinates(X)
```

```
Length(X)
```

The number of coordinate functions of the ambient space of the scheme X. This is equal to the number of coordinates of any point of X.

```
Lengths(X)
```

The lengths of the groups of ones in the gradings of a scroll X.

```
IsHomogeneous(X,f)
```

Returns `true` if and only if the polynomial f is homogeneous with respect to all of the gradings on the scheme X.

```
Multidegree(X,f)
```

The sequence of homogeneous degrees of the polynomial f with respect to the gradings on the scheme X.

112.2.4 Prelude to Points

Points of schemes are handled in an extremely flexible way: their coordinates need not be elements of the base ring, for instance. We don't discuss the details here but simply show how to create points in ambient spaces and illustrate with an example. This is already enough for non-specialised purposes: intrinsics which take point arguments for computing, say, the tangent space to a curve at a point, can take the underlying point of the ambient space or the point of the curve equally well. Having said that, the later section on points, Section 112.7, should be taken as the definitive reference.

```
A ! [a,b,...]
```

```
A(L) ! [a,b,...]
```

For elements $a, b, \ldots$ in the base ring of the scheme A this creates the set-theoretic point $(a, b, \ldots)$ in the affine case, or $(a : b : \ldots)$ in the projective case. Over a field, the projective point will be normalised so that its final nonzero coordinate entry is 1 (and further analogous normalisations when there are at least two gradings as in the case of surface scrolls). The first constructor can only be used if the sequence contains elements of the base ring of A. The second version of the constructor is the standard one. Using it rather than the first allows the user to specify the ring in which the coefficients are to be considered. See the discussion of point sets in the section below on points.

Example H112E8

We create some points with identical coordinates. They are deemed to be unequal by the equality test, but if what you really care about is their coordinates you can check that those are equal.

```
> k<w> := FiniteField(3^2);
> A := AffineSpace(k,2);
> p := A ! [1,2];
> K := ext< k | 2 >;
> q := A(K) ! [1,2];
> m := hom< k -> k | w^3 >;  // Frobenius
> r := A(m) ! [1,2];
> p eq q;
true
>  p eq r;
>>   p eq r;
     ^
Runtime error in 'eq': Arguments are not compatible
Argument types given: Pt, Pt
>  q eq r;
>> q eq r;
     ^
Runtime error in 'eq': Arguments are not compatible
Argument types given: Pt, Pt
```

In the example above, the first method used for the creation of a point is sufficient if you only want to create a point with coefficients in the base ring. The second and third point creations are more precise: they decree exactly the k-algebra in which the coefficients will lie. One should think of the expression `A ! [1,2]` as merely being a convenient shorthand, analogous to defining a sequence without being explicit about its universe.

`Origin(A)`

The origin of the affine space A.

`Simplex(A)`

The sequence of points of the ambient space A having coordinates $(1, 0, \ldots, 0)$, $\ldots (0, \ldots, 0, 1)$ and $(1, \ldots, 1)$ whether A is affine or projective space.

`Coordinates(p)`

The sequence of ring elements corresponding to the coordinates of the point p.

`p[i]`

`Coordinate(p,i)`

The ith coordinate of the point p.

```
p @ f
```

```
f(p)
```

```
Evaluate(f, p)
```

Evaluate the function f of the function field of the scheme X or its ambient at point p which lies on X.

Example H112E9

Although there are many rings which may appear as function fields in various contexts, their elements can all be evaluated at points. (Section 114.8 discusses the function field of a curve which is being used here only as an example.)

```
> A<x,y> := AffineSpace(Rationals(),2);
> FA<X,Y> := FunctionField(A);
> C := Curve(A,x^3 - y^2 + 3*x);
> FC<u,v> := FunctionField(C);
> p := A ! [1,2];
> q := C ! [1,2];
> f := x/y;
> g := X/Y;
> h := u/v;
> Evaluate(f,p), Evaluate(f,q);
1/2 1/2
> Evaluate(g,p), Evaluate(g,q);
1/2 1/2
> Evaluate(h,q);
1/2
> Evaluate(h,p);
1/2
> Evaluate(h,C!p);
1/2
```

112.3 Constructing Schemes

As shown in the examples in the introduction to this chapter, schemes are defined inside some ambient space, either affine or projective space, by a collection of polynomials from the coordinate ring associated with that space. Schemes may also be defined inside other schemes using polynomials from the coordinate ring of the bigger scheme or polynomials from the ambient space.

There is very little difference between creation methods for affine and projective schemes. Of course, in the projective case, the defining polynomials are checked for homogeneity or if an ideal is used, a check is made that its basis contains only homogeneous elements. Otherwise, the only check made at the time of creation is that the polynomials used to define the scheme really do lie in, or are coerced automatically into, the coordinate ring of the chosen ambient space.

Saturation:

As mentioned in the introduction, for schemes in projective spaces, there is a largest ideal which defines that scheme. Technically speaking, if I is any homogeneous defining ideal, this maximal one can be obtained from I by removing the primary components whose radical contains a certain *redundant ideal* of the ambient coordinate ring. This redundant ideal defines sets of points that are illegal projectively. For example, in ordinary projective space, there is one illegal point with all coordinates 0 and this is defined by the redundant ideal $(x_1, \dots, x_n)$. The operation to remove these primary components is ideal saturation of I by the redundant ideal, so we refer to the maximal defining ideal as *saturated* and a scheme X as saturated if its current ideal (as returned by `DefiningIdeal(X)`) is known to be the maximal one.

There are several basic functions that rely on the defining ideal of a scheme X being saturated. The most important are `IsReduced, IsIrreducible, Prime/Primary Components, eq` for any projective schemes and `Dimension, f(X)` (map images) for multi-graded projective schemes.

As the process of saturation may be quite expensive in higher dimensional ambient spaces, the ideal of X is not saturated until the saturation property is required and once saturation has been performed, this is recorded internally. Additionally, scheme constructions like `Union` will automatically produce a result marked as saturated if that can be deduced from the construction method and the saturation state of the argument schemes. In particular, any ambient or scheme defined by a single equation in an ambient is marked as saturated on construction. The `ProjectiveClosure` of an affine scheme is also saturated by construction.

Furthermore, for all of the basic `Scheme` and `Curve` constructors where saturation of the ideal generated by the defining equations is not automatic, there is a `Saturated` parameter that the user can set to be `true` to mark the initial defining ideal as saturated without further checking.

`Scheme(X,f)`

`Scheme(X,F)`

`Scheme(X,I)`

`Scheme(X,Q)`

`Nonsingular`	BoolElt	*Default :* `false`
`Reduced`	BoolElt	*Default :* `false`
`Irreducible`	BoolElt	*Default :* `false`
`GeometricallyIrreducible`	BoolElt	*Default :* `false`
`Saturated`	BoolElt	*Default :* `false`

Create the scheme inside the scheme X defined by the vanishing of the polynomial f, or the sequence of polynomials F, or the ideal of polynomials I, or the ideal in the denominator of the quotient ring $Q = R/I$. In each case, the polynomials must be elements of the coordinate ring of A or automatically coercible into it.

If any of the optional parameters are set to `true`, MAGMA will assume without checking that the scheme has the corresponding property. This may enable subsequent calculations to be done faster; note that if the assumption is not correct, *arbitrary misbehaviour may result.* The option `Saturated` only makes sense when the ambient is projective, and refers to the defining ideal rather than the scheme.

`Cluster(X,f)`

`Cluster(X,F)`

`Cluster(X,I)`

`Cluster(X,Q)`

`Saturated` BOOLELT *Default :* `false`

Create the 0-dimensional scheme inside the scheme X defined by the vanishing of the given polynomials. These can be given as the single polynomial f, or the sequence of polynomials F, or the ideal of polynomials I, or the ideal in the denominator of the quotient ring $Q = R/I$. In each case, the polynomials must be elements of the coordinate ring of X or automatically coercible into it.

Example H112E10

In this example we simply create three schemes. The first is an ambient space A, the affine plane, while the others are subschemes of A.

```
> A<x,y,z> := AffineSpace(Rationals(),3);
> X := Scheme(A,x-y);
> X;
Scheme over Rational Field defined by
x - y
> Y := Scheme(X,[x^2 - z^3,y^3 - z^4]);
> Y;
Scheme over Rational Field defined by
x^2 - z^3
y^3 - z^4
x - y
> Ambient(Y) eq A;
true
```

Note that since Y was created as a subscheme of X it inherits the equations of X. The ambient space of Y is still considered to be A.

`Spec(R)`

The scheme Spec(R) associated to the affine algebra R. A new affine space `Spec(Generic(R))` will be created as the ambient space of this scheme.

`Proj(R)`

The scheme Proj(R) associated to the affine algebra R which will be interpreted with its grading (which will be the standard grading by degree if no other has been assigned). A new projective space `Proj(Generic(R))` will be created as the ambient space of this scheme.

`EmptyScheme(X)`

`EmptySubscheme(X)`

The subscheme of X defined, for an affine scheme X by the trivial polynomial 1, or by maximal ideal $(x_1, \ldots, x_n)$ for a projective scheme X. The returned scheme is marked as saturated.

`X meet Y`

`Intersection(X,Y)`

The intersection of schemes X and Y in their common ambient space. This simply concatenates their defining equations without testing for emptiness.

`X join Y`

`Union(X,Y)`

The union of schemes X and Y in their common ambient space. This is formed by creating the intersection of their defining ideals which is done using a Gröbner basis computation. If both X and Y are saturated then the result is as well and is marked as such.

`&join S`

The union of the schemes in the sequence S in their common ambient space.

`Difference(X, Y)`

Returns the scheme that is obtained by taking the closure of the result of removing (`X meet Y`) from the scheme X, counting multiplicity. The ideal of the result will be the colon ideal of the ideal of X and the ideal of the scheme Y. If X is saturated then the result is as well and is marked as such.

`RemoveLinearRelations(X)`

Convenience function that takes linear relations between variables on X and uses them to eliminate variables. The intrinsic is currently only avaialable for X in ordinary projective space. The result is scheme Y that lies in a lower dimensional projective space that can be identified with the smallest linear subspace of the ambient of X that contains X. Y is returned along with the (linear) scheme isomorphism from X to Y.

Example H112E11

The behaviour of `Difference` is shown.

```
> A2<x,y>:=AffineSpace(Rationals(),2);
> C:=Scheme(A2,(x*y)); //union of the x- and y-axis
> X2:=Scheme(A2,x^2);  //y-axis with double multiplicity
> Difference(X2,C); //y-axis with mult. 1.
Scheme over Rational Field defined by
x
> O:=Scheme(A2,[x,y]);
> Difference(C,O);
Scheme over Rational Field defined by
x*y
```

Removing "ambient" spaces is tricky: Everything is removed.

```
> Difference(C,A2);
Scheme over Rational Field defined by
1
> A3<x,y,z>:= AffineSpace(Rationals(),3);
> C:=Scheme(A3,Ideal([x,z])*Ideal([y,z])); //again, union of x- and y-axis
> Z:=Scheme(A3,[z]); //the x,y plane
> Difference(C,Z);
Scheme over Rational Field defined by
x,
y,
z
```

As one can see, the Z-plane is removed with multiplicities: all that's left is the origin, which has multiplicity 2 in C and only multiplicity 1 in Z.

`Saturate(~X)`

If the scheme X is projective and is not already saturated, saturate its defining ideal.

`AssignNames(~X,N)`

Assign the strings in the sequence N to the ambient coordinate functions of the scheme X.

`X . i`

`Name(X,i)`

The ith coordinate function of the ambient space of the scheme X. The dot notation `X.i` may also be used.

112.4 Different Types of Scheme

As discussed briefly in Section 112.1.7, there are a number of different increasingly specialised data types for schemes. It is often useful to check whether a given scheme can be thought of as belonging to one of these more specialised classes, and if so and appropriate then actually making the type change. In this section we document a number of such type-checking and type-change intrinsics, most of which are of the form `IsSpecialisedType`. These intrinsics always return a boolean value. If that value is `true` then they may also return a new scheme of the given specialised type, although in some trivial cases this does not happen. Of course, each of the different types of scheme has its own methods of construction independently of these intrinsics.

```
IsAffine(X)
```

Returns `true` if and only if the scheme X is an affine space.

```
IsProjective(X)
```

Returns `true` if and only if the scheme X is a projective space. Projective space here includes the case of scrolls.

```
IsOrdinaryProjectiveSpace(X)
```

Returns `true` if and only if the scheme X is a projective space in the usual sense: its coordinate ring has a single grading in which all the variables have weight one.

```
IsAmbient(X)
```

Return `true` if the scheme X is an ambient space.

```
IsCluster(X)
```

Returns `true` if and only if the scheme X is a zero-dimensional scheme (but not the empty scheme). See Section 112.8 for intrinsics which apply to clusters.

```
IsCurve(X)
```

Returns `true` if and only if X is a one-dimensional scheme. See Chapter 114 for intrinsics which apply to curves.

```
IsPlaneCurve(X)
```

Returns `true` if and only if X is a one-dimensional scheme defined by a single equation in a two-dimensional ambient space.

```
IsConic(X)
```

Returns `true` if and only if the scheme X is a curve (in the sense of `IsCurve(X)`) which is nonsingular and defined by an equation of degree 2. See Chapter 119 for intrinsics which apply to such conics.

```
IsRationalCurve(X)
```

Returns `true` if and only if the scheme X is a curve (in the sense of `IsCurve(X)`) which has genus 0. See Chapter 119 for intrinsics which apply to rational curves.

```
IsHyperellipticCurve(X)
```

Return `true` and a hyperelliptic curve if the scheme X is *trivially* a hyperelliptic curve. This occurs if and only if X is already of `CrvHyp` type or is defined by a non-singular Weierstrass equation in correctly weighted two-dimensional projective space. For a general scheme of type `Crv`, the intrinsic `IsHyperelliptic` in Chapter 114 will determine whether X is isomorphic to a hyperelliptic curve and return an isomorphism to a `CrvHyp` if so. This takes a lot more work in general. See Chapter 125 for more information on hyperelliptic curves.

```
IsModularCurve(X)
```

Return `true` if and only if the scheme X is a curve of type `CrvMod`. See Chapter 128 for more information on modular curves.

112.5 Basic Attributes of Schemes

These intrinsics report on basic features of the ambient space of a scheme or the equations defining a scheme. In many cases they simply call the corresponding function of the ambient space; the intrinsic `BaseRing()` is an example. The first set of these functions consists of those that only make reference to the ambient space, while the second set is concerned with the defining equations of the scheme.

112.5.1 Functions of the Ambient Space

```
AmbientSpace(X)
Ambient(X)
```

The ambient space containing the scheme X.

```
SuperScheme(X)
```

The scheme X was created as a subscheme of.

```
BaseRing(X)
CoefficientRing(X)
```

The base ring of the scheme X.

```
BaseField(X)
CoefficientField(X)
```

The base ring of the scheme X if it is a field, otherwise an error.

`IsAffine(X)`

Returns true if and only if the ambient space of the scheme X is affine.

`IsProjective(X)`

Returns true if and only if the ambient space of the scheme X is projective.

`IsOrdinaryProjective(X)`

Returns true if and only if the ambient space of the scheme X is an ordinary projective space, that is, its coordinate ring is generated in degree 1 with respect to the grading on the space.

`IsPlanar(X)`

Return true if the ambient of the scheme X is 2-dimensional.

`IsSaturated(X)`

Returns true if and only if the current defining ideal of the scheme X, as returned by DefiningIdeal(X) is saturated (see section 112.3 on page 3490).

112.5.2 Functions of the Equations

There are many ways to recover the equations which define a scheme. The standard method is to use the DefiningPolynomials function (or its singular versions) since it doesn't involve ideal theory overheads and certainly won't call any Gröbner basis functions.

`DefiningPolynomials(X)`

The defining polynomials for the ideal of the scheme X.

`DefiningPolynomial(X)`

The defining polynomial of the scheme X if it is a hypersurface. If X is not a hypersurface, an error is reported.

`DefiningIdeal(X)`

The ideal of a multivariate polynomial ring defining the scheme X.

`CoordinateRing(X)`

The quotient of the coordinate ring of the ambient space of the scheme X by the ideal of X.

`Curve(X)`

The smallest scheme in the inclusion chain above the scheme X which is a curve.

`GroebnerBasis(X)`

Return a sequence containing the polynomials of a Gröbner basis of the defining ideal of the scheme X. Note that the defining polynomials of X will not be changed, but that the basis of the ideal of X will be updated with the Gröbner basis as is the standard in the multivariate polynomial ring module.

```
MinimalBasis(X)
```

Return a minimal basis of the defining ideal of the scheme X, that is, a sequence of polynomials, no subsequence of which forms a basis of the ideal of X. Note that the defining polynomials of X will not be changed. This is the best human readable basis that Magma can supply.

```
IsHypersurface(X)
```

Returns `true` if and only if the scheme X is definable by a single polynomial. This function will perform a GCD calculation to simplify multiple defining polynomial if possible. The polynomial is returned as a second value.

```
JacobianIdeal(X)
```

The ideal of partial derivatives of the polynomials which define the scheme X.

```
JacobianMatrix(X)
```

The matrix $(\partial f_i/\partial x_j)$ of partial derivatives of the defining polynomials of the scheme X.

```
HessianMatrix(X)
```

The hessian matrix $(\partial^2 f/\partial x_i \partial x_j)$ of the hypersurface X where f is the polynomial which defines X.

```
X eq Y
```

Returns `true` if the schemes X and Y have the same types, ambients and ideals. If Gröbner basis calculations are not available this question may not be able to be decided. If X and Y are projective then they are saturated before ideal equality is tested for.

```
IsSubscheme(X, Y)
```

Returns `true` if and only if the scheme X is contained, scheme-theoretically, in the scheme Y. A Gröbner basis calculation checks the reverse inclusion of the corresponding ideals. If X and Y are projective, then X is saturated before the test for inclusion.

```
IsLinear(X)
```

Return `true` if the scheme X is defined by linear equations, possibly after taking a Gröbner basis.

Example H112E12

In this example we first create some schemes and then test them for inclusions and equality.

```
> P<u,v,w> := ProjectiveSpace(GF(11),2);
> C := Scheme(P,u^2 + u*w + 6*v^2);
> Z := Scheme(C,[u,v]);
> IsSubscheme(Z,C);
true
```

Now we will make another scheme which has the same polynomials as C but which is written in disguise. While the disguise in this case is simply to multiply the polynomial by 2 — the rather-too-obvious false nose and eyebrows among polynomials — the point is to note that the equality test in Magma is not fooled. The equality test identifies that the underlying defining ideals are the same and returns `true`.

```
> D := Scheme(P,2*u^2 + 2*u*w + v^2);
> D eq C;
true
> IsSubscheme(C,D) and IsSubscheme(D,C);
true
> DefiningIdeal(D) eq DefiningIdeal(C);
true
> DefiningPolynomial(D) eq DefiningPolynomial(C);
false
```

As we see in the final line above, checking the equality of ideals corresponds to the natural interpretation of equality.
There are a couple of caveats to this lesson, however. For instance, it is necessary, that the ideals to be comparable, i.e. the schemes must be embedded in the same ambient space.

```
> X<r,s,t> := ProjectiveSpace(GF(11),2);
> E := Scheme(P,r^2 + r*s + 6*t^2);
> E eq C;
false
```

112.6 Function Fields and their Elements

Since the function field of an irreducible variety is a birational invariant, function fields in Magma are associated with the projectively closed varieties. For an affine scheme X, the fields returned by `FunctionField(X)` and `FunctionField(ProjectiveClosure(X))` are identical. Currently, only function fields of projective spaces (and therefore of affine spaces) and curves are supported.

The following functions are provided for working with function fields and their elements. Some of these functions are concerned with converting function field elements into elements of the field of fractions of the coordinate ring of (an affine patch of) the scheme of the function field.

Additionally, function field elements may be used in the definition of scheme maps (see Section 112.14) from the projective or affine schemes on which they are defined to other schemes and may be evaluated at points as described earlier.

Function fields of schemes have type `FldFunFracSch` and their elements have type `FldFunFracSchElt`. These types inherit from `RngFunFracSch`, `RngFunFrac`, `RngMPolRes` and `RngFunFracSchElt`, `RngFunFracElt`, `RngMPolResElt` respectively.

```
Scheme(F)
```

Return the (projective) scheme F is the function field of.

```
IntegerRing(F)
Integers(F)
```

The integer ring of the function field F. This will be the coordinate ring of one of the patches of the scheme of F.

```
AssignNames(~F, S)
```

Assign the strings in S to be the names of the integer ring of F.

```
F ! g
```

Coerce the element g into the function field F of a scheme where g is some function on the scheme of F, for example, g may be an element of the field of fractions of the coordinate ring of a scheme having F as its function field.

```
F . i
```

Return the ith indeterminate of the coordinate ring of the scheme of F as an element of the function field F.

```
ProjectiveFunction(f)
```

Given an element f of a function field of a scheme, return f as an element of the field of fractions of the coordinate ring of the scheme f is a function on.

```
ProjectiveRationalFunction(f)
```

Given an element f of a function field of a (projective) scheme X, returns an element of the field of fractions of the coordinate ring of the ambient of X whose restriction to X as a rational function is f.

```
RestrictionToPatch(f, Xi)
```

Given an element f of a function field of a (projective) scheme X return f as an element of the field of fractions of the coordinate ring of the scheme Xi which must be a patch of X.

`Numerator(f)`

`Denominator(f)`

Given an element f of a function field of a scheme, return the numerator or denominator of f.

`f * g` `f + g` `f - g` `- f` `f / g` `f ^ n`

`f eq g` `IsZero(f)` `IsOne(f)` `IsMinusOne(f)` `IsUnit(f)`

`IntegralSplit(f, X)`

Given a function f on the (projective) scheme X return the numerator and the denominator of g, where g is some rational function on the ambient P of X restricting to f and considered as an element of the field of fractions of the coordinate ring of P.

`Numerator(f, X)`

The first return value of `IntegralSplit(f, X)`.

`Denominator(f, X)`

The second return value of `IntegralSplit(f, X)`.

Example H112E13

Some conversion of function field elements are shown.

```
> P2<X, Y, Z>:=ProjectiveSpace(Rationals(), 2);
> C := Curve(P2, X^2+Y^2-Z^2);
> K<xK, yK> := FunctionField(C);
> aC1<x1,y1> := AffinePatch(C, 1);
> aC2<x2,z2> := AffinePatch(C, 2);
> aC3<y3,z3> := AffinePatch(C, 3);
>
> f := (xK + yK)/(yK);
> K!f;
(xK + yK)/yK
> ProjectiveFunction($1);
(X + Y)/Y
> IntegralSplit(f, C);
X + Y
Y
> RestrictionToPatch(f, aC1);
($.1 + $.2)/$.2
> IntegralSplit(f, aC1);
x1 + y1
y1
> IntegralSplit(f, P2);
x2 + 1
```

```
1
```

`Restriction(f, Y)`

Given f in the function field of the scheme X and Y a subscheme of X with a function field, returns g in the function field of Y obtained by restricting f to Y. If f has a pole along Y, then `Infinity` is returned. An error occurs if f is not defined along Y. Presently, the only nontrivial application of this routine is when Y is a curve and X is the ambient of Y.

`GenericPoint(X)`

Returns a point in the pointset `X(FunctionField(X))` of the scheme X, whose coordinates generate `FunctionField(X)`.

112.7 Rational Points and Point Sets

There are two ways to think of points. If X is a scheme defined over a ring k and L is a k-algebra, then there is a set, called a *point set* and denoted $X(L)$ which is the set of points of X having coordinates in L or, in Magma terminology, the *parent* of such points. Note that in Magma a k-algebra is interpreted to mean any ring which admits coercion from k or which is the codomain of a ring homomorphism whose domain is k. When thinking of points as a sequence of coordinates on some scheme this type of point should be used. It is created by coercing the sequence of coordinates into the required point set using a statement such as

```
> X(L) ! [1,2,3];
```

Alternatively, if the universe of the sequence is equal to the base ring of the scheme, one may simply coerce the sequence into the scheme.

```
> X ! [1,2,3];
```

When the universe of the sequence is the integers, Magma will coerce them into the base ring of the scheme and again this shorthand will work.

The word point always refers to an object whose parent is some point set.

When a scheme is defined over a finite field, there are intrinsics which list all of its points defined over that field or over any finite extension of it.

An alternative approach is to consider points, or sets of points, as schemes in their own right. They can be defined by equations, after all. We call such zero-dimensional schemes *clusters*. They are more general than simply collections of points since their ideals could be nonradical. They are discussed in the Section 112.8 together with intrinsics which translate between points and clusters.

If p is a point, there are two ways of accessing its coordinates. The intrinsic `Coordinates` returns the sequence of all coordinates of p while `p[i]` returns the i-th coordinate alone. For example,

```
> p := X ! [1,2,3];
> Coordinates(p);
[ 1, 2, 3 ]
> p[1];
1
```

See Section 112.2.4 for descriptions of these and some other basic functions.

```
X(L)
PointSet(X,L)
X(m)
PointSet(X,m)
```

The point set of the scheme X of points whose coordinates lie in the ring L or in the codomain of the map m. The map m is a ring homomorphism from the base ring of X to some other ring. Coercion from the base ring of X to L must be possible if m is not given.

```
P eq Q
```

Returns `true` if and only if the point sets P and Q where created on the same scheme and with the same map from the base ring of that scheme.

```
Scheme(P)
```

The scheme X associated to the point set P where P is of the form $X(L)$ for some extension L of the base ring of X.

```
Curve(P)
```

The smallest scheme in the inclusion chain above the scheme associated to the point set P which is a curve.

```
Ring(P)
```

The ring L associated to the point set P where P is of the form $X(L)$ for some scheme X.

```
RingMap(P)
```

The map from the base ring of the scheme of P to the ring of the pointset P.

```
X ! Q
X(L) ! Q
```

The point of the scheme X or the point set $X(L)$ (where X is a scheme and L is some extension ring of its base ring) determined by the sequence of coordinates Q. The universe of the sequence Q must be the base ring of X, or the ring L or some ring from which coercion into one of these is possible.

`p eq q`

Returns `true` if and only if the points p and q lie in some common scheme (possibly after coercion) and their coordinates are equal.

`p in X`

Returns `true` if and only if the point p lies in the scheme X or is coercible into it.

`Scheme(p)`

The scheme on which the point p lies.

`Curve(p)`

The smallest scheme in the inclusion chain above the scheme on which the point p lies which is a curve.

`Q in X`

Returns `true` if and only if all of the points of the set or sequence Q lie in the scheme X or are coercible into it.

`S subset X`

Returns `true` if and only if all points of the set S lie in the scheme X or are coercible into it.

`IsCoercible(X,Q)`

Returns `true` if and only if the sequence Q is the sequence of coordinates of some point of the scheme X. In that case, also return the point.

`RationalPoints(X)`

`RationalPoints(X,L)`

`Points(X)`

`Points(X,L)`

`Bound` RngIntElt *Default :* 1000

An indexed set containing points in the point set $X(L)$, where L is an extension of the base field of X. When not specified, L is taken to be the base field of X. This is implemented in the following situations: (i) L is a finite field, (ii) X has dimension zero, (iii) L is `Rationals()`. In cases (i) and (ii), all the points in $X(L)$ are found. In case (iii), a call to `PointSearch` is made, which searches for points with height up to the specified `Bound` (but note that it does not guarantee finding all of them).

In most cases, the first step is to determine the dimension of X by computing the Groebner basis of its defining ideal. This may be time-consuming; to avoid this one may directly call the relevant search: `Ratpoints` over finite fields, or `PointSearch` (specifying the `Dimension`) over the `Rationals()`.

`RationalPointsByFibration(X)`

`UseHypersurface` BoolElt *Default :* `false`

This is one of the methods used by `RationalPoints` when X is an affine or ordinary projective scheme over a finite field.

The basic idea is to work with a Noether Normalisation of the coordinate ring of X which, in the affine or projective case, gives an everywhere defined map with finite fibres to a linear subspace. We then run over all the points in the subspace adding in the points of X in the finite fibre. Determining the points on a zero-dimensional scheme is relatively fast and so we get a fairly efficient method for listing all points.

There is a variant to the basic algorithm. Rather than running over the linear subspace, it can take fibrations over a hypersurface of dimension one bigger the linear subspace. The points on these as fibred over the subspace may be quicker to find than for a general finite fibration and the finite fibres over the hypersurface are of smaller degree. For example, the general fibre contains only one point when the extra hypersurface equation generates X generically over the subspace.

However it is often slower to use the hypersurface because of the extra computation at the start and the two-stage processing so the default for `UseHypersurface` is `false`. The user may set this parameter to `true` for the variant to be applied.

`Random(S)`

Returns a random point in the pointset $S = X(k)$ where X is a scheme defined over a finite field, and k is a finite field. An error results if the pointset is empty. (Here 'random' simply means all points can occur, but not with uniform distribution.)

This is implemented using the Noether normalisation of X (similarly to `RationalPointsByFibration`).

`HasNonsingularPoint(X)`

`HasNonsingularPoint(X,L)`

Return `true` if and only if the scheme X defined over a finite field contains a nonsingular point (defined over the finite field L if it appears as a second argument). In that case, also return such a point.

Example H112E14

In this example we define a scheme over a finite field and compute some points on it. Note that there are two point constructors used here. The first is simply `X ! Q` where Q is a sequence of integers (or base ring elements). In the second, we try to coerce a sequence Q whose elements do not lie in the base ring. The coercion into X cannot be used here. Instead one must be explicit about the intended point set. We have used the `IsCoercible(X,Q)` intrinsic which creates the point as a side-effect. One could also use the `X(L) ! Q` coercion to create the same point.

```
> A<x,y> := AffineSpace(FiniteField(7),2);
> X := Scheme(A,x^2 + y^2 + 1);
> X ! [2,3];
(2, 3)
```

```
> L<w> := ext< BaseRing(X) | 2 >;
> IsCoercible(X,[w^4,w^4]);
false
> IsCoercible(X(L),[w^4,w^4]);
true (w^4, w^4)
```

Finding those points was not simply good luck. In fact, we worked backwards and computed all points over the base field or L and chose one from each of those sets.

```
> RationalPoints(X);
{@ (3, 2), (4, 2), (2, 3), (5, 3), (2, 4), (5, 4), (3, 5), (4, 5) @}
> #RationalPoints(X,L);
48
```

We now consider an example of a curve in projective 3-space. In older versions of Magma this ran very slowly indeed. Now however, finding points over a fibration, it only takes a few seconds on a fast machine.

```
> k := GF(7823);
> R<x,y,z,w> := PolynomialRing(k, 4);
> I := ideal<R | 4*x*z + 2*x*w + y^2 + 4*y*w + 7821*z^2 + 7820*w^2,
>  4*x^2 + 4*x*y + 7821*x*w + 7822*y^2 + 7821*y*w +
>  7821*z^2 + 7819*z*w + 7820*w^2>;
> C := Curve(Proj(R), I);
> // a genus 0 curve with 1 cusp as singularities => 7823+1 points
> pts := RationalPointsByFibration(C); // could also just use RationalPoints
> #pts;
7824
```

Note that Magma has very fast machinery for computations like this for elliptic and hyperelliptic curves.

112.8 Zero-dimensional Schemes

This section describes intrinsics for creating zero dimensional schemes or *clusters*. It also discusses those functions which converts a finite set of points into the reduced zero dimensional having this support. Throughout this subsection, a lowercase p denotes a point of a scheme.

The word cluster refers to schemes that are known to be zero dimensional. In general, the intrinsic Cluster converts points to clusters while the function RationalPoints finds the points on a cluster which are rational over its base field.

Note that there are four constructors of the form Cluster(X,data) analogous to the four Scheme(X,data) constructors but which make an additional dimension test and type change before returning a cluster determined as a subscheme of X by the data of the second argument.

```
Cluster(p)
Cluster(X, p)
Cluster(S)
Cluster(X, S)
```

The reduced scheme supported at the point p, or supported at the set of points S, as a subscheme of the scheme X if given.

```
RationalPoints(Z)
RationalPoints(Z,L)
```

The set of rational points of the cluster Z. If an extension of the base field L is given as a second argument, the set of points of $Z(L)$, those points whose coordinates lie in L, is returned.

```
PointsOverSplittingField(Z)
```

If Z is a cluster this will determine some (not necessarily optimal) point set $Z(L)$ in which all points of Z having coordinates in an algebraic closure of the base field lie and will return all points of $Z(L)$.

```
HasPointsOverExtension(X)
HasPointsOverExtension(X,L)
```

Returns `false` if and only if all points in the support of the scheme X over an algebraic closure of its base field are already defined over its current base field, or all lie in the point set $X(L)$ if the second argument L is given. This intrinsic is most useful when trying to decide whether or not to make an extension of the base field of X to reveal non-rational points. The base field of X does not need to be a finite field.

```
Degree(Z)
```

The degree of the cluster Z. If Z is reduced, this is equal to the maximum number of points in the support over Z over some extension of its base ring.

Example H112E15

In this example we intersect a pair of plane curves. (Note that much more specialised machinery for working with curves is available in Chapter 114.) First we define two curves and find their points of intersection over the base field. The degree of the cluster Z is the usual numerical *intersection number* of the curves C and D. Here we are more interested in finding exactly those points that lie in the intersection.

```
> k := FiniteField(5);
> P<x,y,z> := ProjectiveSpace(k,2);
> C := Scheme(P,x^3 + y^3 - z^3);
> D := Scheme(P,x^2 + y^2 - z^2);
> Z := Intersection(C,D);
> IsCluster(Z);
true
> Degree(Z);
6
> RationalPoints(Z);
{ (1 : 0 : 1), (0 : 1 : 1) }
> HasPointsOverExtension(Z);
true
```

If C and D were rather general, that is, if Z was reduced, then we would expect 6 points in their intersection. We can't expect that here, but the final line above does confirm that we haven't yet seen all the points of intersection. We allow Magma to compute directly over a splitting field.

```
> PointsOverSplittingField(Z);
{ (0 : 1 : 1), ($.1^14 : $.1^22 : 1), ($.1^22 : $.1^14 : 1), (1 : 0 : 1) }
> L<w> := Ring(Universe($1));
> L;
Finite field of size 5^2
> PointsOverSplittingField(Z);
{ (0 : 1 : 1), (w^14 : w^22 : 1), (w^22 : w^14 : 1), (1 : 0 : 1) }
```

In this case we see that the support is not six points but only four.

112.9 Local Geometry of Schemes

We now discuss intrinsics which apply to a scheme at a single point. At the expense of increasing the number of intrinsics, we try to follow the convention that an intrinsic may simply take a point as its argument or it may take both a point and a scheme as its arguments. In the former case, the implicit scheme argument is taken to be the scheme associated to the point set of the point. In the latter case, it is first checked that the point can be coerced into some point set of the given scheme argument. There are reasons for allowing both methods. Of course, if one is confident about which scheme, X say, a point p lies on then there is no ambiguity about writing, say, `IsNonsingular(p)` rather than `IsNonsingular(X,p)`. On the other hand, the second expression is easier to read, and also guards against the possibility of accidentally referring to the wrong scheme; that is a particular risk here since the answer makes sense even if p lies on some other scheme—imagine the confusion that could arise given a point of a nonsingular curve lying on a singular surface inside a nonsingular ambient space, for instance. But also there are trivial cases when scheme arguments are necessary, `IntersectionNumber(C,D,p)` for example. In fact, that particular example exemplifies the value of points being highly coercible—it is very convenient that the point p could lie in a point set of either C or D or indeed neither of these as long as it could be coerced to them if necessary.

Sometimes a function will require that the point argument is rational, that is, has coordinates in the base ring.

112.9.1 Point Conditions

```
IsSingular(X,p)
```

Returns `true` if and only if the point p is a singular point of the scheme X.

```
IsNonsingular(X,p)
```

Returns `true` if and only if the point p is a nonsingular point of the scheme X.

```
IsOrdinarySingularity(X,p)
```

Returns `true` if and only if the tangent cone to the scheme X at the point p is reduced and X is singular at p. Currently, the scheme X must be a hypersurface.

112.9.2 Point Computations

```
Multiplicity(p)
```

```
Multiplicity(X,p)
```

The multiplicity of the point p as a point of the scheme X. If X is not a hypersurface, computed using local Groebner bases.

```
TangentSpace(p)
```

```
TangentSpace(X,p)
```

The tangent space to the scheme X at the point p. This linear space is embedded as a scheme in the same ambient space as X. An error will be signalled if p is a singular point of X or is not a rational point of X.

```
TangentCone(p)
```

```
TangentCone(X,p)
```

The tangent cone to X at the point p embedded as a scheme in the same ambient space. If the scheme X is not a hypersurface, the computation uses local Groebner bases.

112.9.3 Analytically Hypersurface Singularities

We will say that an isolated singular point p in a pointset $X(k)$ over a field k is a *hypersurface singularity* if the completion of its local ring is isomorphic to the quotient of a power series ring $k[[x_1, \ldots, x_d]]$ by a single power series $F(x_1, \ldots, x_d)$. That is, it is analytically equivalent to the singularity of an analytic hypersurface defined by F at the origin. Clearly d is equal to the dimension of the tangent space at p here. This type of singularity occurs quite commonly (e.g., singular points on actual hypersurfaces, A-D-E singularities on surfaces).

This section contains intrinsics to test whether such a p is a hypersurface singularity and compute the equivalent analytic equation F to given precision, to increase the precision of F at a later stage and to expand a rational function on X to the corresponding element in the field of fractions of $k[[x_1, \ldots, x_d]]$ to any required precision.

```
IsHypersurfaceSingularity(p,prec)
```

The point p is a singular point in $X(k)$ for a field k. It should be an isolated singularity on X and should only lie on irreducible components of X whose dimension d is maximal, i.e. the dimension of X (these conditions are not checked). The integer *prec* should be positive.

The function returns whether p is a hypersurface singularity on X as defined above. This is also equivalent to the conditions that the tangent space of p has dimension $d+1$ and that the local ring at p is a local complete intersection ring.

If this is true, the function also returns extra values. The second return value is a multivariate polynomial F_1 in a polynomial ring $k[x_1,\dots,x_{d+1}]$ such that the singularity p is analytically equivalent to that of the analytic hypersurface $F \in k[[x_1,\dots,x_{d+1}]]$ at the origin and F_1 is equal to F for terms of degree less than or equal to *prec*.

The third is a sequence of simple rational functions on X/k (i.e., quotients of polynomials in the coordinate ring of the ambient of X that, if X is projective, have the same degree for all gradings) such that x_i corresponds to the ith rational function of the sequence. If X is affine, these rational functions will all be k-linear forms in the coordinate variables and if X is ordinary projective, linear forms in the coordinate variables divided by a particular variable that is non-zero at p.

The fourth return value is a data record that is needed if the user wants to later expand F to higher precision or to expand arbitrary rational functions on X/k at p.

The implementation makes use of Magma's local Groebner bases, after localising p to an affine patch and translating it to the origin.

```
HypersurfaceSingularityExpandFurther(dat,prec,R)
```

The record *dat* should be the data record returned for a hypersurface singularity p in $X(k)$ by the intrinsic above, *prec* a positive integer and R a polynomial ring over k of rank $d+1$ (d is the dimension of X).

Returns a polynomial that expands the equation of the analytic hypersurface F to include all terms of degree less than or equal to *prec*. The result will be returned as an element of R (with the ith variable of R corresponding to the analytic variable denoted x_i in the exposition of the previous intrinsic).

This intrinsic is very useful to expand F to higher precision after the original call that determined that p was a hypersurface singularity.

```
HypersurfaceSingularityExpandFunction(dat,f,prec,R)
```

The record *dat* should be the data record returned for a hypersurface singularity p in $X(k)$ by the first intrinsic of this subsection, *prec* a positive integer and R a polynomial ring over k of rank $d+1$ (d is the dimension of X). f should be a rational function on X/k given as an element of the field of fractions of the coordinate ring of the ambient of X or the base change of X to k, if k isn't the base ring of X. f can in fact be given as an element of any rational function field over k whose rank is equal to the rank of the coordinate ring of X. In any case, when X is projective, the numerator and denominator of f have to be homogeneous and have the same degree with repect to all gradings of X.

Returns f pulled back to the analytic coordinate ring at p (identified with $k[[x_1, \ldots, x_{d+1}]]/(F)$ in the notation introduced above) and expanded to required precision. In fact, the return value is given as two polynomials a and b in R, whose variables are identified with the x_i, such that a/b is the finite approximation to the value of the pullback. a and b are actually just the pullbacks of the numerator and denominator of f expanded to include all terms of degree less than or equal to *prec*. The value is returned as two polynomials rather than a quotient as b may be zero if *prec* is sufficiently small even when the denominator of f doesn't vanish on X.

Example H112E16

We consider a singular degree 4 Del Pezzo surface in $\mathbf{P}^4$ over $\mathbf{Q}$ with two conjugate singular points defined over a quadratic extension.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> X := Scheme(P4,[x^2+y^2-2*z^2, x*t+t^2-y*u+2*u^2]);
> IsIrreducible(X);
true
> sngs := SingularSubscheme(X);
> Support(sngs);
> pts := PointsOverSplittingField(sngs);
> pts;
{@ (0 : 0 : 0 : r1 : 1), (0 : 0 : 0 : r2 : 1) @}
> pt := pts[1];
> k := Ring(Parent(pt));
> k;
Algebraically closed field with 2 variables over Rational Field
Defining relations:
[
    r2^2 + 2,
    r1^2 + 2
]
> p := X(k)!Eltseq(pt);
> boo,F,seq,dat := IsHypersurfaceSingularity(p,3);
> boo;
true
> R<a,b,c> := Parent(F);
> F;
2*r1*a*b^2 - 2*a^2*c - 3/2*r1*a*c^2 + 8*a^2 - 4*b^2 + 4*r1*a*c + c^2
> HypersurfaceSingularityExpandFurther(dat,4,R);
-a^4 - r1*a^3*c + 1/2*a^2*c^2 + 2*r1*a*b^2 - 2*a^2*c - 3/2*r1*a*c^2 +
Ψ8*a^2 - 4*b^2 + 4*r1*a*c + c^2
> Rk<p,q,r,s,w> := PolynomialRing(k,5);
>  //can use Rk as base-changed coordinate ring here
> HypersurfaceSingularityExpandFunction(dat,(p^2+q*s)/w^2,3,R);
2*r1*a^3 - 3*a^2*c - 1/2*r1*a*c^2 + 4*a^2 + (2*r1 + 1)*a*c - 1/2*c^2 + r1*c
```

1

112.10 Global Geometry of Schemes

Many of the names of intrinsics in this section come from the usual terminology of algebraic geometry. A reference for them is Hartshorne's book [Har77], especially Chapter II, Section 3.

`Dimension(X)`

The dimension of the scheme X. If X is irreducible then the meaning of this is clear, but in general it returns only the dimension of the highest dimensional component of X. The dimension of an empty scheme will be returned as -1. If the dimension is not already known, a Gröbner basis calculation is employed. If X is projective in a multi-graded ambient then it is saturated before this calculation takes place. The computation method involves computing the `Dimension` of the `Ideal` of the scheme, and then (for projective schemes) subtracting the number of gradings.

`Codimension(X)`

The codimension of the scheme X in its ambient space. In fact, this number is calculated as the difference of `Dimension(A)` and `Dimension(X)` where A is the ambient space, so if X is not irreducible this number is the codimension of a highest dimensional component of X.

`Degree(X)`

The degree of the scheme X.

`ArithmeticGenus(X)`

The arithmetic genus of a scheme X. The ambient space of X must be ordinary projective space.

`IsEmpty(X)`

Returns `true` if and only if the scheme X has no points over any algebraic closure of its base field. This intrinsic tests if the ideal of X is trivial (in a sense to be interpreted separately according to whether X is affine or projective) and then applies the Nullstellensatz.

`IsNonsingular(X)`

Returns `true` if and only if the scheme X is nonsingular and equidimensional over an algebraic closure of its base field. The test `IsEmpty` for the emptiness of the scheme is applied to the scheme defined by the vanishing of appropriately sized minors of the jacobian matrix of X.

`IsSingular(X)`

Returns `true` if and only if the scheme X either has a singular point or fails to be equidimensional over an algebraic closure of its base field.

`SingularSubscheme(X)`

The subscheme of the scheme X defined by the vanishing of the appropriately sized minors of the jacobian matrix of X. If X is not equidimensional, its lower dimensional components will be contained in this scheme whether they are singular or not.

`PrimeComponents(X)`

A sequence containing the irredundant prime components of the scheme X.

`PrimaryComponents(X)`

A sequence containing the irredundant primary components of the scheme X.

`ReducedSubscheme(X)`

The subscheme of X with reduced scheme structure, followed by the map of schemes to X. This function uses a Gröbner basis to compute the radical of the defining ideal of X.

`IsIrreducible(X)`

Returns `true` if and only if the scheme X has a unique prime component. If X is not a hypersurface, a Gröbner basis calculation is necessary and X is saturated before this occurs if it is projective.

`IsReduced(X)`

Returns `true` if and only if the defining ideal of the scheme X equals its radical. If X is a hypersurface the evaluation of this intrinsic uses only derivatives so works more generally than the situations where a Gröbner basis calculation is necessary. In the latter case, X is saturated before the calculation if it is projective.

`IsCohenMacaulay(X)`

`IsGorenstein(X)`

`IsArithmeticallyCohenMacaulay(X)`

`IsArithmeticallyGorenstein(X)`

`CheckEqui` BoolElt *Default :* `false`

These intrinsics currently only apply to schemes in ordinary projective space. The first two intrinsics return whether X is (locally) Cohen-Macaulay/Gorenstein, meaning that the local ring of every the scheme-theoretic point on X satisfies the property. The second two intrinsics return whether the coordinate ring of X (the polynomial coordinate ring of the projective ambient quotiented by the maximal defining ideal of X) satisfies the corresponding property. The arithmetic version implies the local version. The results are stored internally with X for future reference. Also, if X is known to be non-singular, we can immediately deduce the local version of the properties is true and this check is also performed internally.

There is a further slight restriction in that X has to be equidimensional (each irreducible component having the same dimension and there being no other scheme-theoretic "associated points" beside the generic points of the irreducible components : true if X is also reduced). This is *not* checked by default in order to save some computation time. If the user is unsure whether X is equidimensional, the `CheckEqui` parameter should be set to `true` which forces a check.

The implementations use the minimal free polynomial resolution of the maximal defining ideal of X. The arithmetic versions are actually faster than the plain versions. `IsGorenstein` may be particularly heavy computationally as it has to check whether the canonical sheaf is locally free of rank 1 after the Cohen-Macaulay property has been verified.

Example H112E17

In this example we write down a rather unpleasant scheme and analyse the basic properties of its components.

```
> A<x,y,z> := AffineSpace(Rationals(),3);
> X := Scheme(A,[x*y^3,x^3*z]);
> Dimension(X);
2
> IsReduced(X);
false
> PrimaryComponents(X);
[
    Scheme over Rational Field defined by
    x,
    Scheme over Rational Field defined by
    x^3
    y^3,
    Scheme over Rational Field defined by
    y^3
    z
]
> ReducedSubscheme(X);
Scheme over Rational Field defined by
x*y
x*z
```

The reduced scheme of X is clearly the union of a line and a plane. The scheme X itself is more complicated, having another line embedded in the plane component.

112.11 Base Change for Schemes

Let A be some ambient space in Magma. For example, think of A as being the affine plane. Let k be its base ring and R_A its coordinate ring. If $m : k \to L$ is a map of rings (a coercion map, for instance) then there is a new ambient space denoted A_L and called the *base change of A to L* which has coordinate ring R_{A_L} with coefficient ring L instead of k. (Mathematically, one simply tensors R_A with L over k. In Magma the equivalent function at the level of polynomial rings is `ChangeRing`.) There is a base change function described below which takes A and L (or the map $k \to L$) as arguments and creates this new space A_L. Note that there is a map from the coordinate ring of A to that of A_L determined by the map m.

This operation is called *base extension* since one often thinks of the map m as being an extension of fields. Of course, the map m could be many other things. One key example where the name *extension* is a little unusual would be when m is the map from the integers to some finite field.

Now let X be a scheme in Magma. Thus X is defined by some polynomials $f_1, \ldots, f_r$ on some ambient space A. Given a ring map $k \to L$ there is a base change operation for X which returns the *base change of X to L*, denoted X_L. This is done by first making the base change of A to L and then using the map from the coordinate ring of A to that of A_L to translate the polynomials f_i into polynomials defined on A_L. These polynomials can then be used to define a scheme in A_L. It is this resulting scheme which is the base change of X to L.

If one has a number of schemes in the same ambient space and wants to base change them all at the same time, a little care is required. The function which takes a scheme and a map of rings as argument will create a new ambient space each time so is unsuitable. Better would be to base change the ambient space and then use the base change function which takes the scheme and the desired new ambient space as argument. (This latter base change function appears to be different from the other ones. In fact it is not. We described base change above as a function of maps of rings. Of course, there is a natural extension to maps of schemes. With that extension, this final base change intrinsic really is base change with respect to map of ambient spaces.)

```
BaseChange(A,K)
```

```
BaseExtend(A,K)
```

If A is a scheme defined over a field k and K is an extension into which elements of k can be automatically coerced then this returns a new scheme A_K defined over K. No cached data about A will be transferred to A_K and coordinate names will have to be defined again on A_K if needed.

```
BaseChange(A,m)
```

```
BaseExtend(A,m)
```

If m is a map of rings whose domain is the base ring of the scheme A, this returns the base change of A to the codomain of m. The equations of A, if any, are mapped to the new ambient coordinate ring using m.

```
BaseChange(F,K)
BaseExtend(F,K)
BaseChange(F,m)
BaseExtend(F,m)
```

If F is a sequence of schemes lying in a common ambient space whose base ring admits automatic coercion to K or is the domain of a ring map m then this returns the base change of the elements of F as a new sequence.

```
BaseChange(X,A)
BaseExtend(X,A)
BaseChange(X,A,m)
BaseExtend(X,A,m)
```

If X is any scheme whose ambient space B is of the same type (affine or projective) and dimension as the ambient space A but either has a base ring which admits coercion to that of A or the map m is a ring map from the base ring of B to that of A then this returns a scheme with the equations of X as a subscheme of A. The equations are transferred to A using coercion or the map m.

```
BaseChange(X, n)
BaseExtend(X, n)
```

The base change of the scheme X, where the base ring of X is a finite field to the finite field which is a degree n extension of the base field of X.

Example H112E18

Here are two curves whose intersection points are not defined over the rationals and one of which only splits after a field extension. The basic function to calculate intersection points only searches for them over the current field of definition so misses them at first. But with an extra argument it is able to search over an extension of the base without actually changing base of the schemes. This contrasts with finding higher dimensional components of schemes which always requires the base change.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,x^2 + y^2);
> IsIrreducible(C);
true
> D := Curve(A,x - 1);
> IntersectionPoints(C,D);
{}
> Qi<i> := QuadraticField(-1);
> IntersectionPoints(C,D,Qi);
{ (0, i), (0, -i) }
```

So we have found the intersection points (although we haven't explained how we chose the right field extension). Now we do the same calculation again but by making the base change of all schemes to the field Qi. Over this field the intersection points are immediately visible, but also the curve C splits into two components.

```
> B<u,v> := BaseChange(A,Qi);
> C1 := BaseChange(C,B);
> D1 := BaseChange(D,B);
> IsIrreducible(C1);
false
> IntersectionPoints(C1,D1);
{ (0, i), (0, -i) }
> PrimeComponents(C1);
[
    Scheme over Qi defined by u + i*v,
    Scheme over Qi defined by u - i*v
]
```

112.12 Affine Patches and Projective Closure

In Magma, any affine ambient space A has a unique projective closure. This may be assigned different variable names just like any projective space. The projective closure intrinsics applied to affine schemes in A will return projective schemes in the projective closure of A. Conversely, a projective space has a number of standard affine patches. These will be the ambient spaces of the standard affine patches of a projective scheme. In this way, the closures of any two schemes lying in the same space will also lie in the same space. The same goes for standard affine patches. These relationships between affine and projective objects are very tightly fixed: asking for the projective closure of an affine scheme will always return the identical object, for instance.

```
ProjectiveClosure(X)
```

The projective closure of the scheme X. If the projective closure has already been computed, this scheme will be returned. If X is an affine space for which no projective closure has been computed, the projective closure will be a projective space with this space as its first standard patch. Otherwise, the result will lie in the projective closure of the ambient space of X. If X has been computed as an affine patch the projective closure will be the scheme it is an affine patch of even if this is not mathematically correct (see Example H112E20).

```
AffinePatch(X,i)
```

The ith affine patch of the scheme X. The number of affine patches is dependent on the type of projective ambient space in which X lies, but for instance, the standard projective space of dimension n has $n+1$ affine patches. In that case, i can be any integer in the range $1, \dots, n+1$. The order for affine patches is the natural one once you decide that the first patch is that with final coordinate entry nonzero (in the projective closure).

`AffinePatch(X,p)`

A standard affine patch of the scheme X containing the point p. The second return value is the point corresponding to p in that patch.

`IsStandardAffinePatch(A)`

Return whether the affine space A is a standard affine patch of its projective closure and if so which patch it is. For A to be a non–standard patch means that its projective closure must have been set using `MakePCMap`. Returns `false` if A does not have a projective closure to be a patch of.

`NumberOfAffinePatches(X)`

Return the number of standard affine patches of the scheme X (O if X is an affine scheme).

`HasAffinePatch(X, i)`

Return whether the ith patch of the scheme X can be created.

Example H112E19

This example shows that taking patches and closures several times really does return identical schemes.

```
> A1<u,v> := AffineSpace(GF(5),2);
> X := Scheme(A1,u^2 - v^5);
> PX<U,V,W> := ProjectiveClosure(X);
> PX;
Scheme over GF(5) defined by
U^2*W^3 + 4*V^5
> AffinePatch(PX,1) eq X;
true
> X2<u2,w2> := AffinePatch(PX,2);
> X2;
Scheme over GF(5) defined by
u2^2*w2^3 + 4
> ProjectiveClosure(X2) eq ProjectiveClosure(X);
true
```

Example H112E20

Even if those schemes are not mathematically correct.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> L := Curve(P2,Z);
> Laff := AffinePatch(L,1);
> Dimension(Laff);
-1
> Laff;
```

```
Scheme over Rational Field defined by
1
> ProjectiveClosure(Laff) eq L;
true
> ProjectiveClosure(EmptyScheme(Ambient(Laff)));
Scheme over Rational Field defined by
1
```

`HyperplaneAtInfinity(X)`

The hyperplane complement of the scheme X in its projective closure.

`ProjectiveClosureMap(A)`

`PCMap(A)`

The map from the affine space A to its projective closure.

`AffineDecomposition(P)`

Projective spaces have a standard disjoint decomposition into affine pieces—not the same thing as the affine patches—of the form

$$\mathbf{P}^n = \mathbf{A}^n \cup \mathbf{A}^{n-1} \cup \cdots \cup \mathbf{A}^1 \cup p$$

where $\mathbf{A}^n$ is the first affine patch, $\mathbf{A}^{n-1}$ is the first affine patch on the hyperplane at infinity and so on. Finally, p is the point $(1 : 0 : \cdots : 0)$. This intrinsic returns a sequence of maps from affine spaces to the projective space P whose images are these affine pieces of a decomposition. The point p is returned as a second value.

`CentredAffinePatch(S, p)`

An affine patch of S centred at the point p and the embedding into S, achieved by translation of a standard affine patch.

112.13 Arithmetic Properties of Schemes and Points

This section contains several functions of arithmetic interest, that apply to general schemes defined over the rationals, number fields, or function fields.

112.13.1 Height

```
HeightOnAmbient(P)
```

```
Absolute        BoolElt        Default : false
Precision       RngIntElt      Default : 30
```

The height of the given point, as a point in the ambient projective space (or affine space). This is the *exponential* height (for the logarithmic height, take `Log`!). By default it is the *relative* height, unless the optional parameter `Absolute` is given. The function works when the ambient space (of the scheme containing P) is any affine or projective space (possibly weighted), and the ring of definition of P is contained in the rationals, a number field or a function field.

Note that `Height` is also defined for certain special schemes, such as elliptic and hyperelliptic curves. In these cases the `Height` is a *canonical, logarithmic, absolute* height.

112.13.2 Restriction of Scalars

```
RestrictionOfScalars(S, F)
WeilRestriction(S, F)
```

```
SubfieldMap         Map           Default :
ExtensionBasis      [ FldElt ]    Default : []
```

Given an affine scheme S whose base ring is a field K, the function returns its restriction of scalars from K to F. The scheme S must be contained in an affine space. The field F should be either a subfield of K, or else isomorphic to a subfield of K; in this case the inclusion map may be specified as the optional argument `SubfieldMap`, or otherwise is the same as the map returned by `IsSubfield(F,K)`.

The restriction of scalars S_{res} is a scheme over F satisfying the following functorial property (for point sets): $S_{res}(R) = S(R \otimes_F K)$ for all rings $R \supseteq F$.

Four objects are returned; the first is the scheme S_{res}, and the other three are maps of various kinds:

- the natural map of schemes from the base extension $(S_{res})_{\otimes K}$ to S,
- a function which takes a point in a point set $S_{res}(R)$ and computes its image under the map $S_{res}(R) \rightarrow S(RK)$ (if there is no known relationship between R and K an error results), and
- a map of point sets $S(K) \rightarrow S_{res}(F)$.

The result is obtained by direct substitution using the standard basis of K/F, or the basis given in ExtensionBasis if this is specified.

112.13.3 Local Solubility

Let X be a scheme over a number field K. One of the fundamental problems in arithmetic geometry is to decide if $X(K)$ is empty. In general, this is a very hard question. One way to arrive at an affirmative answer is to prove that $X(L)$ is empty for some larger field L. An important class of such field is formed by *complete local fields*. These are obtained by taking the p-adic topology on K associated to a prime ideal p of the ring of integers of K and to consider the topological completion L of K (see `Completion` on page 3-889).

```
IsEmpty(Xm)
```

`Smooth`	BoolElt	*Default :* `false`
`AssumeIrreducible`	BoolElt	*Default :* `false`
`AssumeNonsingular`	BoolElt	*Default :* `false`
Verbose	`LocSol`	*Maximum :* 2

Let X be a scheme over a number field K and let L be a completion of K at a finite prime. If *Xm* is the point set of X taking values in L, this function returns if the point set is empty. If `false` is returned, then (a minimal approximation to) a point is returned.

Setting `AssumeIrreducible := true` tells the system to assume that X is irreducible. This leads to unpredictable results if the components of X have distinct dimensions.

Setting `AssumeNonsingular := true` tells the system to assume that X has no L-rational singular points. If there are such points and this flag is set, then an infinite loop can result.

Setting `Smooth := true` tells the system to only consider nonsingular points on X. This is only implemented for plane curves. The system will blow up any L-rational singular points when encountered and test the desingularized model for solubility. If a point is found, then an approximation may be returned that is indistinguishable from a singular point.

The following classes of schemes are recognized and treated separately :

(i) Hyperelliptic curves over fields with large residue field of odd characteristic. For these fields a generalization by Nils Bruin of an algorithm presented in [MSS96] is used. The complexity of this algorithm is independent of the size of the residue field.

(ii) Hyperelliptic curves over fields with small residue field or residue field of even characteristic. For these fields a depth-first backtracking algorithm is used to construct a solution.

(iii) Nonsingular curves represented by a possibly singular planar model. In this case, a depth-first backtracking algorithm is used to construct a solution. Whenever a tentative solution approximates a rational singularity, that singularity is blown up and the construction is continued on the desingularized model with Hensel's lemma as a stopping criterion. Use `Smooth := true` to access this option.

(iv) An intersection in $\mathbf{P}^3$ of a quadratic cone with a singularity at $(0:0:0:1)$ and another quadric. This case (often needed in computations concerning elliptic curves) is handled by testing hyperelliptic curves for local solubility.

(v) General schemes. The system decomposes X into primary components over K, which are equidimensional. For each of the components, it tests the singular subscheme for solubility. If a solution is found, this is returned. Otherwise, it tests the scheme itself for solubility using a depth-first backtracking algorithm with Hensel's lemma as stopping criterion.

Note that to construct a point set over a completion of a number field, one should use `PointSet(X,phi)` where `Kp, phi := Completion(K,p)`. This rather cryptic syntax is due to the fact that, although a completion strictly speaking is an *extension* of K, Magma does not recognise this fact and therefore does not allow for automatic coercion into K_p. The system has to be presented explicitly with the map ϕ from K into K_p.

For a description of the algorithms used, see [Bru04].

Example H112E21

Some usage of `IsEmpty` is illustrated below.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2,X^2+Y^2);
> IsEmpty(C(pAdicField(2,20)));
false (O(2^2) : O(2^2) : 1 + O(2^20))
> IsEmpty(C(pAdicField(2,20)):Smooth);
true
> K<i> := NumberField(PolynomialRing(Rationals())![1,0,1]);
> CK := BaseChange(C,K);
> p := (1+i)*IntegerRing(K);
> Kp,toKp := Completion(K,p);
> CKp := PointSet(CK,toKp);
> IsEmpty(CKp:Smooth);
false (O(Kp.1^3) : O(Kp.1^3) : 1 + O(Kp.1^100))
```

Example H112E22

And a simpler example.

```
> p := 32003;
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(p*(x^10+p*x^3-p^2*4710));
> Qp := pAdicField(p);
> time IsEmpty(C(Qp));
true
Time: 0.090
```

`IsLocallySolvable(X, p)`

`Smooth`	BoolElt	*Default :* `false`
`AssumeIrreducible`	BoolElt	*Default :* `false`
`AssumeNonsingular`	BoolElt	*Default :* `false`

Given a projective scheme X defined over a number field or over the rationals, test if the scheme is locally solvable at the prime ideal p (for number fields) or prime number p (for rationals) indicated. If the scheme is found to have a local point, then `true` is returned together with an approximation to a point. Otherwise, `false` is returned.

The optional parameters have the same meaning as for `IsEmpty` (see above).

For a description of the algorithms used, see [Bru04].

Example H112E23

A locally solvable scheme and a non–locally solvable scheme (when considering only non–singular points) are both shown below.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2,X^2+Y^2);
> IsLocallySolvable(C,2);
true (O(2^2) : O(2^2) : 1 + O(2^50))
> IsLocallySolvable(C,2:Smooth);
false
> K<i>:=NumberField(PolynomialRing(Rationals())![1,0,1]);
> CK:=BaseChange(C,K);
> p:=(1+i)*IntegerRing(K);
> IsLocallySolvable(BaseChange(C,K),p:Smooth);
true (O($.1^3) : O($.1^3) : 1 + O($.1^100))
```

`LiftPoint(P, n)`

`Strict`	BoolElt	*Default :* `true`

`LiftPoint(F, d, P, n)`

`Strict`	BoolElt	*Default :* `false`

Let P be in $X(L)$, where X is a scheme over the rationals or over a number field and L is a completion of the base field at a finite prime. This routine attempts to lift P to the desired precision n using quadratic newton iteration. This routine only works if P is distinguishable from all singular points on X. Note that if X is of positive dimension, then the lift is inherently arbitrary.

Due to limited precision used in the computation, it may be impossible to attain the desired precision exactly. By default this leads to an error. If the user specifies `Strict := false` the system silently returns a lift with maximum attainable precision if the desired precision cannot be reached.

The second form provides the same functionality, but requires all input data to be supplied over L. The sequence F should be the defining equations of a scheme over L of dimension d. The sequence P should be coordinates of an approximation of a point on that scheme. The returned sequence is a lift of the point described by P to precision n, if possible.

In principle, a point returned by `IsEmpty` has sufficient precision for `LiftPoint` to work. However, `IsEmpty` may perform nontrivial operations on the scheme. A nonsingular point on a component of X may be singular on X itself.

Example H112E24

A lift of a point in a non empty pointset is given.

```
> P2<X,Y,Z>:=ProjectiveSpace(Rationals(),2);
> C:=Curve(P2,X^2+Y^2-17*Z^2);
> Qp:=pAdicField(2,20);
> bl,P:=IsEmpty(C(Qp));
> LiftPoint(P,15);
(9961 + O(2^15) : O(2^15) : 1 + O(2^20))
```

112.13.4 Searching for Points

The following intrinsic implements a nontrivial method to search for points on any scheme defined over the rationals.

`PointSearch(S,H : `*parameters*`)`

`Dimension`	RngIntElt	*Default :* 0
`Primes`	SeqEnum	*Default :* [ ]
`OnlyOne`	BoolElt	*Default :* `false`

Searches for points on the scheme S up to roughly height H. The scheme must be in either an affine space or a non-weighted projective space.

This uses a p-adic algorithm: first find points locally modulo a small prime (or two small primes), then lift these p-adically, and then see if these give global solutions. Lattice reduction is used at this stage, and this makes the method far more efficient than a naive search, for most schemes. Note that points which reduce to singular points modulo p are not necessarily found.

If `OnlyOne` is `true`, then the computation will terminate as soon as one point is found. The algorithm computes the dimension of the scheme unless `Dimension` is set to a nonzero value.

The algorithm chooses its own primes unless `Primes` is non-empty; either one prime or two can be specified. The algorithm slows down considerably in higher dimension — for threefolds, it can take a few hours to search for points up to height 500 or so. For special types of curves (ternary cubics, intersection of quadrics), efficient search methods are implemented under other names (for instance `Points` and `PointsQI`).

Example H112E25

```
> P<a,b,c,d> := ProjectiveSpace(Rationals(),3);
> S := Scheme (P, a^2*c^2 - b*d^3 + 2*a^2*b*c + a*b^3 - a*b^2*c +
>                7*a*c^2*d + 4*a*b*d^2);
> Dimension(S);
2
> time PS := PointSearch(S,100);
Time: 9.050
> #PS; // not necessarily exhaustive
67571
```

112.14 Maps between Schemes

Given schemes X and Y one can define a map $f : X \to Y$ in a number of ways. The basic method is to give a sequence of polynomials or quotients of polynomials defined on X. If X has an associated function field then function field elements may also be used. Alternative sets of defining polynomials/rational functions may also be given, as long as these represent the same rational map as the original defining set.

Scheme maps in Magma represent *rational* maps between schemes. That is, a map $f : X \to Y$ between schemes X and Y actually corresponds to a morphism from a dense Zariski-open subset of X to Y and two maps f and g from X to Y are considered to be equal (and will be so deemed by `eq`, for example) if they are equal as morphisms when restricted to a dense open subset of X that lies in a domain of definition of f and a domain of definition of g. The precise open domain of definition U of a map f is unspecified but in most functional contexts, it is equal to the complement of the base scheme f as returned by `BaseScheme(f)`. This is the set of scheme-theoretic points at which none of the maps given by the defining polynomials/rational functions or any set of alternative defining polynomials are 'naively' defined. `Extend` computes alternative defining equations that reduces the base scheme so that its open complement is equal to the maximal domain of definition of the rational map represented. However, this is a fairly generic implementation that can be computationally very heavy in many cases. When the domain X is a curve with function field and the codomain Y is ordinary projective, the implicit domain of definition for computing the image of points is the maximal domain of definition of f rather than the base scheme. The function field machinery is used here.

Similarly, isomorphisms between schemes can be defined using inverse defining polynomials/rational functions and these represent birational maps rather than actual scheme isomorphisms.

There are some natural functions associated to a map f and some other more complicated functions. The most natural things are, for a point p of X, computing the image point $f(p)$ and, for a subscheme $S \subset Y$ computing the preimage scheme `Pullback(f,S)`. More complicated functions include the standard Gröbner basis algorithm for computing images $f(T) \subset Y$ of subschemes T of X. This is defined when T doesn't lie in the base scheme of f and the image is actually the scheme theoretic closure of the image of f restricted to the open subset of T on which it is defined.

For some functions such as the image computation just mentioned, it is a requirement that the schemes X and Y be defined over a common base ring and that the map f be defined over the identity map (or coercion map) of this common base ring.

Maps respect point set structures. Indeed, given a map f as above and an extension L of k, the base ring of X, there is a map $f(L) : X(L) \to Y(L)$. The point set of Y is determined by the composition of the map of base rings with the extension map $k \to L$. This map $f(L)$ cannot be created without creating the scheme map f first, and in any case it is not usually explicitly required: the evaluation of $f(p)$ will invoke it in the background, for example. But when the main purpose of f is to transfer a large number of points from $X(L)$ to $Y(L)$ then it is best to assign $g := f(L)$ explicitly and use the map g. In this way the small additional overheads involved in repetitively constructing $f(L)$ are avoided. Perhaps more importantly, it also ensures that the image points all lie in the same point set.

Maps respect projective closures of schemes. That is, given f as above one can compute the projective closure of f which is a map from the projective closure of X to that of Y and which agrees with f where they are both defined.

From V2.16, there is a slight variant on the basic scheme map type (`MapSch`), which we refer to as scheme graph maps and are defined by the graph of the map in the product of the domain and codomain. These currently only have very basic constructors and somewhat less functionality than scheme maps. They will be described in a separate subsection.

Much of the functionality only works for scheme maps f between schemes whose base rings are fields. We usually do not explicitly mention in this documentation when this is a requirement, but the functions will return an error if the base rings are not fields in cases where they must be.

112.14.1 Creation of Maps

The most basic map constructors are described here. It is possible to create maps defined over a map of the base rings of the schemes.

```
map< X -> Y | F >
```

```
map< X -> Y | F, G >
```

```
map< X -> Y | u, F >
```

Check	BoolElt	*Default :* true
CheckInverse	BoolElt	*Default :* true

Create the map $X \to Y$ of schemes determined by the sequence F of polynomials or rational functions defined on X. The two schemes X and Y must be defined over compatible base rings, that is, there must exist a ring map u from the base ring of Y to that of X. The polynomials can be elements of the coordinate ring of the ambient space containing X or elements of the coordinate ring of X itself. In any case, the polynomials will be lifted to elements of the coordinate ring of the ambient space and any ring elements which admit this operation could be used. If the function field of X exists then elements of this may also be used.

The argument u is a map from the base ring of Y to that of X. If it is omitted then natural coercion map is assumed. In practice, the base rings are often identical in which case coercion will simply be the identity map.

A birational inverse specified by polynomials or rational functions on Y can be given as the sequence G.

It is also allowed for F and G to be a sequence of sequences of polynomials or rational functions, giving alternate sets of defining equations (inverse defining equations) that must define the same rational map on X (rational inverse on Y). The base scheme of the map (the inverse) is the intersection of the base schemes of the individual sets of defining equations (inverse defining equations) as described in `BaseScheme`.

There are two parameters `Check` and `CheckInverse` which control exactly what validation checks are carried out.

If the parameter `Check` is set to `false` then no checking occurs at all.

If `Check` is `true` but CheckInverse is `false`, then the map is checked to be well defined from domain to codomain and any inverse to be well defined in the opposite direction. However if there are inverse functions specified, it is not checked that the forward and inverse functions actually define (birational) inverse maps.

If both parameters are `true` then all checks mentioned above are done.

```
iso<  X -> Y | F, G  >
```

Create the map $X \to Y$ of schemes determined by the sequence F for which the map $Y \to X$ determined by the sequence G is a birational inverse. The sequences F and G should contain polynomials or rational functions defined on X and Y respectively. The two schemes X and Y must be defined over the same base ring.

The two check parameters for the main map constructor also apply here with the same meaning.

Example H112E26

Map creation is very similar to that for maps between polynomial rings, although here one must put the polynomial arguments into a single sequence.

```
> k := Rationals();
> A<t> := AffineSpace(k,1);
> B<x,y> := AffineSpace(k,2);
> f := map< A -> B | [t^3 + t, t^2 - 3] >;
> f;
Mapping from: Aff: A to Aff: B
with equations :
t^3 + t
```

```
t^2 - 3
```

Features of the map can be computed. Of course, the domain and codomain are trivial attributes of the map while its image requires a Gröbner basis computation.

```
> Domain(f) eq A;
true
Variables : t
> Codomain(f);
Affine Space of dimension 2
Variables : x, y
> Image(f);
Scheme over Rational Field defined by
-x^2 + y^3 + 11*y^2 + 40*y + 48
```

Example H112E27

Defining maps using function field elements directly can be particularly convenient for curves. For example, the user may have a set of such functions coming from some divisor computations. Here is a simple (artificial!) example where a rational map from a projective curve or any of it's affine patchs to the projective line corresponding to a rational function is defined.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^2+y^2-z^2);
> C1 := AffinePatch(C,1);
> C2 := AffinePatch(C,2);
> F := FunctionField(C);
> f := F!(x/y);
> P1 := ProjectiveSpace(Rationals(),1);
> mp1 := map<C->P1 | [f,1]>;
> mp2 := map<C1->P1 | [f,1]>;
> mp3 := map<C2->P1 | [f,1]>;
```

Example H112E28

Maps of the base ring can be included in scheme maps as is the case for maps of polynomial rings. Here we make a Frobenius map.

```
> k<w> := FiniteField(3^2);
> u := hom< k -> k | w^3 >;
> A<t> := AffineSpace(k,1);
> f := map< A -> A | u, [t^3] >;
> f;
Mapping from: Aff: A to Aff: A
with equations :
t^3
and map between base rings
```

```
Mapping from: FldFin: k to FldFin: k given by a rule [no inverse]
```

Notice next how the map f fixes points defined over the prime subfield of k but moves those points with coordinates having nontrivial w component.

```
> p := A ! [w];
> f(p);
(w^3)
> f(A ! [2]);
(2)
```

```
IdentityMap(X)
```

Create the identity map of the scheme X.

```
ConstantMap(X,Y,p)
map<  X -> Y | Q  >
```

The map taking all points of the scheme X to the point p of scheme Y where Q is the sequence of coordinates of p.

```
Projection(X,Y)
```

The linear projection from projective space X to projective space Y that omits the first $\dim X - \dim Y$ coordinates.

```
Projection(X, Q)
Projection(X)
Projection(X, p)
```

The projection of the scheme X away from the point p into the projective ambient space Q (if given). If p is not given it is taken to be $(1 : 0 : \ldots : 0)$.

```
ProjectionFromNonsingularPoint(X,p)
```

The projection of the scheme X from the nonsingular (and rational) point p of X. The projection map is returned as a second value. The image of the blowup of p as a point of X is returned as a third value. If this is a point, it is returned as a point type.

```
ProjectiveMap(L, Y)
ProjectiveMap(L)
```

Given a list L of functions in the function field of X, where X is a projective scheme, return the projective map $X \to Y$ defined by taking those functions as projective coordinates. The scheme Y should be a projective space of dimension one less than the length of L. If Y is not supplied, a new projective space of appropriate dimension is created.

`ProjectiveMap(f, Y)`

`ProjectiveMap(f)`

A short form for `ProjectiveMap([f,1],X,Y)`.

Example H112E29

The above function is illustrated.

```
> P2<X,Y,Z>:=ProjectiveSpace(Rationals(),2);
> C:=Curve(P2,
>      X^4-2*X^3*Y-X^2*Y^2-2*X^2*Y*Z+2*X*Y^3+2*X*Y^2*Z+Y^4-7*Y^3*Z+Y^2*Z^2);
> omega:=CanonicalDivisor(C);
> Degree(omega); //genus 0 curve
-2
> L:=Basis(-omega);
> L;
[
    ($.2 - 4/25) * ($.1^3 - $.1*$.2 + 5/56*$.2^3 + 20/63*$.2^2) * ($.2 - 4) *
    ($.1^3*$.2 - 1/4*$.1^2*$.2^2 - 26/5*$.1^2*$.2 + 4/5*$.1^2 - 7/4*$.1*$.2^3 -
        2*$.1*$.2^2 - 3/4*$.2^4 + 151/20*$.2^3 + 4*$.2^2 - 4/5*$.2)^-1,
    ($.2) * ($.1^2 - 3/14*$.2^2 + 1/63*$.2) * ($.2 - 4) * ($.2 - 4/25) *
    ($.1^3*$.2 - 1/4*$.1^2*$.2^2 - 26/5*$.1^2*$.2 + 4/5*$.1^2 - 7/4*$.1*$.2^3 -
        2*$.1*$.2^2 - 3/4*$.2^4 + 151/20*$.2^3 + 4*$.2^2 - 4/5*$.2)^-1,
    ($.1 + 13/28*$.2 - 8/63) * ($.2 - 4) * ($.2 - 4/25) * ($.2)^2 * ($.1^3*$.2 -
        1/4*$.1^2*$.2^2 - 26/5*$.1^2*$.2 + 4/5*$.1^2 - 7/4*$.1*$.2^3 -
        2*$.1*$.2^2 - 3/4*$.2^4 + 151/20*$.2^3 + 4*$.2^2 - 4/5*$.2)^-1
]
> mp:=ProjectiveMap(L,P2); //anticanonical embedding
> mp;
Mapping from: Crv: C to Prj: P2
with equations :
23/16*X^3 - 113/64*X^2*Y + 5/16*X^2*Z - 71/64*X*Y^2 + 93/16*X*Y*Z + 21/64*Y^3 -
    61/64*Y^2*Z - 5/16*Y*Z^2
-1/8*X^3 + 23/16*X^2*Y + X^2*Z - 19/16*X*Y^2 - 15/8*X*Y*Z - 21/16*Y^3 +
    115/16*Y^2*Z - Y*Z^2
X^3 - 53/32*X^2*Y - 1/8*X^2*Z - 11/32*X*Y^2 - 3/4*X*Y*Z + 21/32*Y^3 -
    13/32*Y^2*Z + 1/8*Y*Z^2
> mp(C);
Curve over Rational Field defined by
X^2 - 15/8*X*Y + 23/4*X*Z - 229/32*Y^2 - 17/16*Y*Z - Z^2
```

`Elimination(X,V)`

The affine scheme obtained by eliminating the ambient variables of the affine scheme X whose indices appear in V from the equations of X. Thus if $V = [2, 5]$ then the result will be a scheme in the affine subspace $u = v = 0$ where u and v are the second and fifth variables of the ambient space of X.

`Inverse(f)`

The inverse of the map of schemes f if inverse defining equations have been included in the definition of f, otherwise an error.

`IsInvertible(f)`

Tests whether the map f between schemes is birational. If so, returns a birational inverse. This function works fairly generically, using Groebner basis computations over standard affine patches of the domain and codomain to compute the closure of the graph of f and retrieve inverse equations from that. It can be very expensive computationally.

`HasKnownInverse(f)`

Returns `true` if the map f has an inverse stored.

Example H112E30

This example shows how `IsInvertible` is more powerful then `Inverse`.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2, X^3*Y^2 + X^3*Z^2 - Z^5);
> Genus(C);
1
> pt := C![1,0,1];
> E,toE := EllipticCurve(C,pt);
> IsInvertible(toE);
true Mapping from: CrvEll: E to Crv: C
with equations :
$.1^3
-3*$.1*$.2*$.3 - 9*$.2*$.3^2
$.1^3 + 3*$.1^2*$.3
and inverse
-3*X^2*Z + 3*X*Z^2
-3*X^2*Y
X^2*Z - 2*X*Z^2 + Z^3
>  Inverse(toE);
>> Inverse(toE);
          ^
Runtime error in 'Inverse': Map has no inverse
```

```
g * f
```

The composition $g \circ f$, but note the convention for order of composition: the order of mapping is that g acts first and is followed by f. Strictly speaking, one might want to see evaluation of points done on the left to make sense of this, (this can be done using `p @ f` instead of `f(p)`). Since one would usually assign a new identifier to this composition, this is not a large problem. Only simple error checking is done — domain–codomain matching and that the composition doesn't have so many zero components that it is projectively illegal.

Where the expansion of such compositions could be expensive, the resulting map will be stored as a composition. The equivalent expanded map can be created by

```
> gf := g*f;
> dp := DefiningPolynomials(gf);
> dpi := InverseDefiningPolynomials(gf);
> m := map<D -> C | dp, dpi>;
```

where D is the domain of g and C is the codomain of f. The composition will act differently to the expanded map - it will be undefined at all the places each factor is undefined.

```
Components(f)
```

The maps composed to form f.

Example H112E31

As part of its generic map machinery, Magma has a structure for the set of all maps between two given schemes. There is also a structure for the group of all automorphisms of a scheme which is discussed in Section 112.14.6. Using this space one can realise the effect of a map on Hom spaces. We will make two Hom spaces having a common codomain.

```
> k := Rationals();
> P<x,y,z,t> := ProjectiveSpace(k,3);
> A := Scheme(P,Minors(M,2))
>           where M is Matrix(CoordinateRing(P),2,3,[x,y,z,y,z,t]);
> B := Scheme(P,x*t - y*z);
> F<r,s,u,v> := RuledSurface(k,0,0);
> HomAF := Maps(A,F);
> HomBF := Maps(B,F);
> HomAF;
Set of all maps from A to F
```

Given a map $A \to B$ we make the map from `HomBF` to `HomAF` given by composition. Although A lies inside B, we choose a map $A \to B$ which isn't this inclusion.

```
> i := map< A -> B | [y,x,t,z] >;
> ii := map< HomBF -> HomAF | g :-> i * g >;
```

The map ii of Hom spaces realises the composition of maps with i. We test this on a single map $f : B \to F$.

```
> f := map< B -> F | [x,y,z,t] >;
> Expand(ii(f)) eq Expand(i*f);
true
```

`Restriction(f,X,Y)`

Check BoolElt *Default :* `true`

The restriction of the map of schemes f to the scheme X in its domain. The codomain of the new map is considered to be the scheme Y which must either contain the codomain of f, or lie in that codomain but contain the scheme $f(X)$.

By default the program checks these subscheme relationships; this may be time-consuming, and can be skipped by setting the optional parameter `Check` to `false`.

`Expand(phi)`

Given a map ϕ between schemes stored in factored form, return the map in expanded form. Note that if ϕ is a composite of maps, each with many alternative defining polynomials, then computing the expansion can be very expensive. In other situations computing the expansion of ϕ can cause huge (intermediate) expressions and therefore be very expensive as well.

Due to the automatic simplification in map creation, the base scheme of the returned map might be smaller than the base scheme of ϕ.

`Extend(phi)`

Given a map ϕ between schemes, returns an expanded map with extra alternative equations in order to reduce the base scheme as far as possible, i.e. so that the open complement of the base scheme is the maximal domain of definition of the rational map represented by ϕ. This routine is potentially very expensive because it requires Groebner basis computations on several affine graph ideals of ϕ.

`Prune(phi)`

Given a map ϕ between schemes in expanded form, removes alternative equations that do not reduce the base scheme of ϕ. If ϕ has a known inverse, this is returned unaltered.

`Normalization(phi)`

`Normalisation(phi)`

The map created from ϕ by removing common factors from the defining polynomial.

Example H112E32

It is shown below how to convert a map between schemes into a proper morphism.

```
> P2<x,y,z>:=ProjectiveSpace(Rationals(),2);
> C:=Curve(P2,x^3+y^3-2*z^3);
> E,phicomp:=EllipticCurve(C,Place(C![1,1,1]));
> Puvw<u,v,w>:=Ambient(E);
```

We get ϕ as phicomp in expanded form.

```
> phi:=Expand(phicomp);
> phi;
Mapping from: Crv: C to CrvEll: E
with equations :
3*x^2*y - 3*x^2*z - 3*x*y^2 + 9*x*y*z - 6*x*z^2 - 6*y^2*z + 15*y*z^2 - 9*z^3
-9*x^2*y + 27*x*y^2 - 54*x*y*z + 18*x*z^2 + 45*y^2*z - 81*y*z^2 + 27*z^3
y^3 - 3*y^2*z + 3*y*z^2 - z^3
```

But ϕ still has a base scheme.

```
> Degree(BaseScheme(phi));
6
```

So we extend ϕ to phiext defined on C entirely.

```
> phiext:=Extend(phi);
> phiext;
Mapping from: Crv: C to CrvEll: E
with equations :
3*x^2*y - 3*x^2*z - 3*x*y^2 + 9*x*y*z - 6*x*z^2 - 6*y^2*z + 15*y*z^2 - 9*z^3
-9*x^2*y + 27*x*y^2 - 54*x*y*z + 18*x*z^2 + 45*y^2*z - 81*y*z^2 + 27*z^3
y^3 - 3*y^2*z + 3*y*z^2 - z^3
and alternative equations :
3*x^3 + 6*x^2*z - 6*x*y*z + 9*x*z^2 + 3*y^3 - 15*y*z^2
-18*x^2*z + 54*x*y*z - 27*x*z^2 + 99*y*z^2
x*z^2 + 2*y^2*z - 3*y*z^2
```

And there is no base scheme anymore!

```
> Degree(BaseScheme(phiext));
0
```

This map is invertible.

```
> bl,phii:=IsInvertible(phiext);
> assert bl;
> phii;
Mapping from: CrvEll: E to Crv: C
with equations :
1/6*u^3 - 13/2*u^2*w - 2*u*v*w + 21/2*u*w^2 - 1/6*v^2*w + 3*v*w^2 + 27*w^3
u^2*w - 30*u*w^2 - 3*v*w^2 + 36*w^3
u^2*w - 3*u*w^2 - 18*w^3
```

```
and inverse
3*x^2*y - 3*x^2*z - 3*x*y^2 + 9*x*y*z - 6*x*z^2 - 6*y^2*z + 15*y*z^2 - 9*z^3
-9*x^2*y + 27*x*y^2 - 54*x*y*z + 18*x*z^2 + 45*y^2*z - 81*y*z^2 + 27*z^3
y^3 - 3*y^2*z + 3*y*z^2 - z^3
and alternative inverse equations :
3*x^3 + 6*x^2*z - 6*x*y*z + 9*x*z^2 + 3*y^3 - 15*y*z^2
-18*x^2*z + 54*x*y*z - 27*x*z^2 + 99*y*z^2
x*z^2 + 2*y^2*z - 3*y*z^2
```

But the inverse still has a base scheme.

```
> Degree(BaseScheme(phii));
6
```

So extend `phii`.

```
> phiiext:=Extend(phii);
```

No base scheme left!

```
> Degree(BaseScheme(phiiext));
0
```

Note that the inverse – `phiext` – is still retained. So `phiiext` now really is a morphism.

```
> phiiext;
Mapping from: CrvEll: E to Crv: C
with equations :
1/6*u^3 - 13/2*u^2*w - 2*u*v*w + 21/2*u*w^2 - 1/6*v^2*w + 3*v*w^2 + 27*w^3
u^2*w - 30*u*w^2 - 3*v*w^2 + 36*w^3
u^2*w - 3*u*w^2 - 18*w^3
and inverse
3*x^2*y - 3*x^2*z - 3*x*y^2 + 9*x*y*z - 6*x*z^2 - 6*y^2*z + 15*y*z^2 - 9*z^3
-9*x^2*y + 27*x*y^2 - 54*x*y*z + 18*x*z^2 + 45*y^2*z - 81*y*z^2 + 27*z^3
y^3 - 3*y^2*z + 3*y*z^2 - z^3
and alternative equations :
-1/234*u^3 + 7/26*u^2*w + 5/117*u*v*w - 57/26*u*w^2 + 1/702*v^2*w - 6/13*v*w^2 -
    27/13*w^3
-1/39*u^2*w + 1/117*u*v*w + 21/13*u*w^2 - 1/351*v^2*w + 2/39*v*w^2 - 54/13*w^3
u*w^2 - 1/351*v^2*w + 4/39*v*w^2 + 18/13*w^3
-729/26*u^3 + 3/13*u^2*v + 32805/26*u^2*w + 17/13*u*v^2 + 4293/13*u*v*w -
    19683/26*u*w^2 + 4/39*v^3 + 537/26*v^2*w - 1377/13*v*w^2 + 19683/13*w^3
-27/13*u^2*v + 9/13*u*v^2 + 1233/13*u*v*w + 4/39*v^3 + 132/13*v^2*w -
    1215/13*v*w^2
u*v^2 + 4/39*v^3 - 9/13*v^2*w + 756/13*v*w^2
and alternative inverse equations :
3*x^3 + 6*x^2*z - 6*x*y*z + 9*x*z^2 + 3*y^3 - 15*y*z^2
-18*x^2*z + 54*x*y*z - 27*x*z^2 + 99*y*z^2
x*z^2 + 2*y^2*z - 3*y*z^2
```

112.14.2 Basic Attributes

112.14.2.1 Trivial Attributes

```
Domain(f)
```

The domain of the map of schemes f.

```
Codomain(f)
```

The codomain of the map of schemes f.

```
DefiningPolynomials(f)
DefiningEquations(f)
```

The sequence of functions used to define the map of schemes f.

If f is stored as an unexpanded composition then it will be expanded and the defining equations of the expansion returned.

```
FactoredDefiningPolynomials(f)
```

If the map of schemes f was created by composition (and not expanded) return the sequence of sequences of the defining equations of the maps which were composed to form f otherwise return `DefiningPolynomials` of f.

```
InverseDefiningPolynomials(f)
```

The sequence of functions used to define the inverse of the map of schemes f.

If f is stored as an unexpanded composition then it will be expanded and the inverse defining equations of the expansion returned.

```
FactoredInverseDefiningPolynomials(f)
```

If the map of schemes f was created by composition (and not expanded) and has an inverse return the sequence of sequences of inverse defining equations of the maps which were composed to form f otherwise return `InverseDefiningPolynomials` of f.

```
AllDefiningPolynomials(f)
```

The polynomials of all definitions of the map of schemes f.

```
AllInverseDefiningPolynomials(f)
```

The polynomials of all definitions of the inverse of the map of schemes f if f has a known inverse.

```
AlgebraMap(f)
```

The underlying map of polynomial rings determining the map of schemes f. Thus if F is the sequence of defining equations of f and x is the first variable of the codomain then `F[1]` will be the image of x under `AlgebraMap(f)`.

```
FunctionDegree(f)
```

The degree of the homogeneous polynomials which define the map f of projective schemes. If there are alternative defining polynomials, returns the minimum value over the different sets of defining polynomials.

112.14.2.2 Basic Tests

```
f eq g
```

Returns `true` if and only if the maps of schemes f and g have the same domain and codomain and define the same rational map.

```
IsRegular(f)
IsPolynomial(f)
```

Returns `true` if and only if the map of schemes f is defined at all points of its domain.

```
IsIsomorphism(f)
```

Returns `true` if and only if the map of schemes $f : X \to Y$ has inverse defining equations or if they may be easily computed (e.g. the projective closure of the map has inverse defining equations). If so, return a map $g : X \to Y$ which is of the recognised isomorphism type as a second value.

```
IsDominant(f)
```

Returns `true` if and only if the closure of the image of the map of schemes f is the whole of its codomain.

```
IsLinear(f)
```

Returns `true` if and only if the map of schemes f is a regular map defined by linear polynomials.

```
IsAffineLinear(f)
```

Returns `true` if and only if the map of schemes f is a map between affine spaces defined by polynomials of degree at most 1.

112.14.3 Maps and Points

Given a map $f : X \to Y$ of schemes and point p of X outside of the base scheme of f, then the image $f(p)$ is a point of Y. Moreover, given an extension K of the base rings of X and Y, there is a map of point sets $f(K) : X(K) \to Y(K)$. This isn't often needed, but should be used as in the example below, when very many point images are required. Note that it will ensure that all points are returned in the same determined point set. Maps also behave well with respect to sets and sequences of points.

```
f(p)
```

The point $f(p)$ if the point p is in a point set of the domain of the map of schemes f and doesn't lie in the base scheme of f. An error results if p is in the base scheme (except in the curve case described below). Sets and sequences of points are handled in the same way. If the domain of the map is a curve C with a function field (see Algebraic Curves chapter) and p is in the base scheme of f, then now (from V2.17), Magma tries to compute the image of p by working with the function field places over p without having to extend f via `Extend`. If p is non-singular then there is only one place above it and the image will exist if the codomain is projective. If there are several places over p, the image will be computed and returned when the image of all of the places is the same and is defined over p's point set base ring (that must be a field).

```
Pullback(f, p)
```

The preimage of a point p under the map of schemes f. When f is an isomorphism of schemes with an inverse g, the returned result is the point $g(p)$. Otherwise the pullback is returned as a subscheme of the domain of f. This is identical to `Pullback(f,S)`, where S is the one-point scheme containing p.

When a scheme is returned, it will contain the base scheme of f (which won't map to p under f), but this can be remedied using the (potentially expensive) function call `Difference(Pullback(f,p), BaseScheme(f))`.

```
p @@ f
```

This is the same as `Pullback(f,p)` *except* when f is an isogeny between elliptic curves. in which case one rational point in the preimage is returned (if none exist, an error results).

```
f(K)
f(m)
```

The map induced by the map of schemes $f : X \to Y$ on point sets $X(K) \to Y(K)$. If m is a ring map from the base ring of X and u is the map of base rings from Y to X then $f(m)$ will be the map of point sets $X(m) \to Y(m(u))$.

Example H112E33

Mapping a single point is easy.

```
> P1<s,t> := ProjectiveSpace(Rationals(),1);
> P3<w,x,y,z> := ProjectiveSpace(Rationals(),3);
> f := map< P1 -> P3 | [s^4,s^3*t,s*t^3,t^4] >;
> p := P1 ! [2,1];
> f(p);
(16 : 8 : 2 : 1)
```

If many points need to be mapped from a fixed point set, a small overhead can be avoided by working with the map of point sets directly.

```
> K := QuadraticField(5);
> g := f(K);
> ims := [];
> for i in [1..100] do
> Append(~ims, g(P1 ! [i,1]));
> end for;
```

This example could also have been handled in one step using a sequence constructor.

```
> pts := [ P1 ! [i,1] : i in [1..100] ];
> f(pts) eq ims;
true
```

An example where the projection from an elliptic curve to the projective line contains the origin in its base scheme but the image under the extension of the projection is computed by using the place machinery without having to globally extend the map.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> E := Curve(P2,Y^2*Z-X^3-X*Z^2);
> p := E![0,0,1];
> f := map<E->P1|[X,Y]>;
> p in BaseScheme(f);
true (0 : 0 : 1)
> f(p);
(0 : 1)
```

112.14.4 Maps and Schemes

The natural operation for maps on schemes is *pullback*, that is, compute the preimage. This is rather trivial. On the other hand, computing the image of schemes under a map requires a Gröbner basis calculation so is much harder. If the map has an inverse this image calculation is automatically replaced by the more simple pullback using the inverse map.

Note that, strictly speaking, the image algorithm computes the closure of the image of the map. We still call it the image, though, and don't worry that there may be some points of the image that are not the set-theoretic image of any point of the domain.

Over a field, the equations of the image of a map in a particular degree can be computed using linear algebra, so a distinct intrinsic is provided for this.

Other schemes related to a map are also discussed here.

```
Pullback(f, X)
```

The scheme in the domain of the map of schemes f given by the pullback of the equations defining the subscheme X of the codomain of f. The result will contain the base scheme of f (which won't map onto X under f) but this can be remedied using the (potentially expensive) function call `Difference(Pullback(f,X), BaseScheme(f))`.

```
Image(f)
```

```
f(X)
```

Let X be a subscheme of the domain of the map of schemes f such that, if U is the open complement of the base scheme of f, $X \cap U$ is Zariski-dense in X. This intrinsic returns the scheme-theoretic closure of $f(X \cap U)$ in the codomain of f, which we refer to simple as the image $f(X)$. For the first signature the image of the entire domain is returned. Moreover, it is stored with the map f so can be called again later without any recomputation. Note that if the domain of f is projective multigraded, then X is saturated before the computation to ensure the correct result. For computaional efficiency, we do not check that $X \cap U$ is indeed Zariski-dense in X.

```
Image(f,X,d)
```

A basis of the polynomials of degree d in the codomain of the map of schemes f which contain the image $f(X)$. The scheme X must be a subscheme of the domain of f and d must be a positive integer.

For best results for projective schemes, remove the contribution of the irrelevant ideal corresponding to the zero point from X (normalize the equations of the scheme).

Example H112E34

Consider the embedding of the projective line in 3-space as a quartic. It can be defined as the image of a map determined by a 4-dimensional subspace of the degree 4 monomials on the line, for instance: see [Har77] Chapter II, Example 7.8.6 or the later section here on linear systems.

```
> P1<s,t> := ProjectiveSpace(Rationals(),1);
> P3<w,x,y,z> := ProjectiveSpace(Rationals(),3);
> f := map< P1 -> P3 | [s^4,s^3*t,s*t^3,t^4] >;
> Image(f);
Scheme over Rational Field defined by
-w^2*y + x^3
w*y^2 - x^2*z
-x*z^2 + y^3
-w*z + x*y
> IsNonsingular(Image(f));
true
> f(p) in Image(f) where p is P1 ! [2,1];
true (16 : 8 : 2 : 1)
```

If the Gröbner basis computation is too expensive, or if a partial solution for the image computation would be acceptable, the function `Image(f,C,d)` described above and illustrated in the next example calculates those hypersurfaces of degree d containing $f(C)$. Given a bound on d, the equations of the image could also be calculated using this function.

Example H112E35

A situation where one is really interested in the equations of the image in a particular degree occurs in the case of canonical curves. Usually the ideal is generated in degree 2, but for trigonal curves the degree 2 generators only cut out a surface scroll on which the curve is cut out by a relative equation of degree 3.
Here we simply assert that the curve C has genus 5 and that the map f is the canonical map of the curve C. Chapter 114 describes functions that determine both invariants.

```
> k := Rationals();
> P2<X,Y,Z> := ProjectiveSpace(k,2);
> P4<a,b,c,d,e> := ProjectiveSpace(k,4);
> C := Curve(P2, X^5 + X*Y^3*Z + Z^5);
> f := map< P2 -> P4 | [Y*Z, X*Y, Z^2, X*Z, X^2] >;
> S := Image(f,C,2);
> S;
Scheme over Rational Field defined by
a*d - b*c
a*e - b*d
c*e - d^2
> Dimension(S);
2
> f(C);
Curve over Rational Field defined by
-d^2 + c*e,
-b*d + a*e,
-b*c + a*d,
a*b^2 + c^2*d + e^3,
a^2*b + c^3 + d*e^2
```

In this case both image computations are fast so the timing difference between them is tiny. But pushing the genus a little higher soon makes the point.
It is easy to see that S is a scroll: its equations are the rank 2 minors of the matrix

$$\begin{pmatrix} a & c & d \\ b & d & e \end{pmatrix}.$$

```
BaseScheme(f)
```

The subscheme of the domain X of the map of schemes f where the map is 'naively' not defined. When f is expanded, this is equal to the intersection of the base schemes of each of the alternate sets of defining polynomials/rational functions. For a single set of defining polynomials/rational functions, the base scheme is defined by the union of subschemes of X where a denominator of a defining rational function vanishes when the codomain is affine, and by the subset of points of X which are mapped by the given defining polynomials into a null point (with coordinates on which all polynomials in an irrelevant ideal vanish) when the codomain is projective.

```
BasePoints(f)
BasePoints(f,L)
```

If the base scheme of the map of schemes f is finite, this returns a sequence containing those points defined over the base ring which lie in it. Otherwise an error is reported. If a second argument L is included which is an extension field of the base field then base points defined over L are returned.

Example H112E36

We find the base points of a map, although we have to extend the field before we find them all.

```
> k := GF(7);
> P<x,y,z> := ProjectiveSpace(k,2);
> p := x^2 + y^2;
> f := map< P -> P | [p*x,p*y,z^2*(z-x)] >;
> BasePoints(f);
{}
> Degree(BaseScheme(f));
6
> HasPointsOverExtension(BaseScheme(f));
true
```

Clearly we are not seeing all the points of indeterminacy of f. We clumsily extend the base field until we do see enough points. Of course, it is clear that the problem is the polynomial p, so a degree 2 extension will be enough.

```
> BasePoints(f,ext<k|2>);
```

```
{ (1 : $.1^36 : 1), ($.1^12 : 1 : 0), ($.1^36 : 1 : 0), (1 : $.1^12 : 1) }
> HasPointsOverExtension(BaseScheme(f),ext<k|2>);
false
```

Example H112E37

In this example we make an elementary transformation of scrolls.

```
> Q := Rationals();
> F<u,v,x,y> := RuledSurface(Q,2);
> G<a,b,r,s> := RuledSurface(Q,3);
> F;
Rational Scroll of dimension 2
Variables : u, v, x, y
Gradings :
1       1       -2      0
0       0       1       1
> phi := map< F -> G | [u,v,x,y*u] >;
```

Next we find the base points of the map ϕ by hand.

```
> Scheme(F,[u,v]) join Scheme(F,[x,u*y]);
Scheme over Rational Field defined by
u*x
v*x
u*y
> RationalPoints($1);
{ (0 : 1 : 0 : 1) }
```

The map ϕ is the elementary transformation in the point $(0:1:0:1)$ of F. That is, it is the blowup of this point followed by the contraction of the birational transform of the fibre through this point.

112.14.5 Maps and Closure

```
ProjectiveClosure(f)
```

The map induced by the map of schemes f between the projective closure of its domain and codomain. If either domain or codomain is already projective, then it remains unchanged in the new map. In particular, if both domain and codomain are already projective, then the returned map is simply f itself.

```
MakeProjectiveClosureMap(A, P, S)
MakePCMap(A, P, S)
MakeProjectiveClosureMap(m)
MakePCMap(m)
```

If A is an affine space and P a projective space and if S is a sequence of polynomials on A defining a map from A to P (or if m is such a map) then this map is set as the projective closure map of A. There is very little functionality for projective closure maps which are not the standard ones, so this intrinsic is usually used in cases where no relationship between A and P yet exists but the user would like A to behave as a standard patch on P so that the closure of a scheme in A is a scheme in P.

```
RestrictionToPatch(f,j)
```

The restriction of the map f, a map of schemes from an affine scheme to a projective scheme, to a rational map from its domain to the jth standard affine patch of its codomain.

```
RestrictionToPatch(f,i,j)
```

The restriction of the map f, a map between two projective schemes, to a rational map from the ith standard affine patch of its domain to the jth patch of its codomain.

Example H112E38

The application of closure and patching functions is straightforward. To compute the restriction of a map f to the ith patch of the domain and jth patch of the codomain, essentially set the $\dim + 2 - i$th coordinate function to 1 and divide by the $\dim + 2 - j$th defining equation of f.

```
> P<w,x,y,z> := ProjectiveSpace(Rationals(),3);
> f := map< P -> P | [1/w,1/x,1/y,1/z] >;
> f12 := RestrictionToPatch(f,1,2);
> f12;
Map of affine spaces defined by [ $.3/$.1, $.3/$.2, $.3 ]
```

The functions are inevitably rational so cannot be expressed in any coordinates that might already exist on the affine patches. Instead they are expressed in terms of the generators of the function fields.

```
> ProjectiveClosure(f12);
Map of projective spaces defined by [ x*y*z, w*y*z, w*x*z, w*x*y ]
> ProjectiveClosure(f12) eq f;
true
```

However, as seen in the final line above, the relationship between a map and its closure is maintained.

112.14.6 Automorphisms

Automorphisms of schemes defined over a field may be constructed. The main cases where there is significant functionality is for automorphisms of affine and projective spaces and curves. Recall that for projective spaces the only regular automorphisms are the linear maps. However there are many more rational automorphisms, often called *Cremona transformations*. In the case of the projective plane, these form a group generated by linear automorphisms together with a single quadratic transformation. In higher dimensions, the structure of this group is unknown.

Affine spaces have much more complicated automorphism groups. Decomposition results are known in the case of the affine plane over certain fields (the complex numbers for instance), but otherwise no general statements are known. More information and references can be found in [vdE00], especially in the opening essay.

Although automorphisms can be computed, groups of automorphisms cannot be computed except in a very few cases. For the case of curves, see the `Algebraic Curves` chapter. For ambients, there is currently a function `AutomorphismGroup` which returns a group together with a map matching group elements with the automorphism they represent only in the case of linear automorphisms of projective spaces defined over a finite field.

`Automorphism(X,F)`

The automorphism of the scheme X determined by the sequence of polynomials F defined on X. This function uses a Gröbner basis calculation. If the inverse functions are already known then one can use the `map< | >` constructor and then a type change or the `iso< | >` constructor. This is illustrated in Example H112E39.

`IdentityAutomorphism(X)`

`IdentityMap(X)`

The identity map $X \to X$ on the scheme X.

`IsEndomorphism(f)`

Returns `true` if and only if the domain and range of the maps of schemes f are equal.

`IsAutomorphism(f)`

Returns `true` if and only if the maps of schemes f is an automorphism of its domain. In this case f is returned as an automorphism as the second value.

Example H112E39

In this example we show how to make an automorphism from equations, and also how to include the equations of the inverse map if they are already known. The automorphism is the hyperelliptic involution on a hyperelliptic curve, although we don't use the machinery of hyperelliptic curves here.

```
> A<u,v> := AffineSpace(Rationals(),2);
> f := v^5 + 2*v^3 + 5;
> C := Curve(A,u^2 - f);
> phi := Automorphism(C,[-u,v]);
> Type(phi);
MapAutSch
> phi;
Mapping from: Crv: C to Crv: C
with equations :
-u
v
and inverse
-u
v
```

In this case we clearly know the inverse map in advance. We can make an automorphism of C as follows.

```
> psi := map< C -> C | [-u,v],[-u,v] >;
> psi eq phi;
true
> Type(psi);
MapSch
```

The map ψ is fine, but it is not of the same type as ϕ. We make the type change, if desired, as follows.

```
> bool,psi1 := IsAutomorphism(psi);
> bool;
true
> Type(psi1);
MapAutSch
```

Example H112E40

A standard Gröbner basis exercise is to test particular examples of the Jacobian conjecture. This states that a polynomial map of the plane is invertible if (and only if) its jacobian determinant is everywhere nonzero. The problem here is to calculate the conjectured inverse polynomial map.

```
> A<u,v> := AffineSpace(Rationals(),2);
> f := u^3 + 3*u^2*v^2 + 3*u^2 + 3*u*v^4 + 6*u*v^2 + v^6 + 3*v^4 + v + 3;
> g := u + v^2 + 1;
> J := JacobianMatrix([f,g]);
> Determinant(J);
```

```
-1
> m := map< A -> A | [f,g] >;
> m;
Mapping from: Aff: A to Aff: A
with equations :
u^3 + 3*u^2*v^2 + 3*u^2 + 3*u*v^4 + 6*u*v^2 + v^6 + 3*v^4 + v + 3
u + v^2 + 1
> IsAutomorphism(m);
true
> m;
Mapping from: Aff: A to Aff: A
with equations :
u^3 + 3*u^2*v^2 + 3*u^2 + 3*u*v^4 + 6*u*v^2 + v^6 + 3*v^4 + v + 3
u + v^2 + 1
and inverse
-u^2 + 2*u*v^3 - 6*u*v + 10*u - v^6 + 6*v^4 - 10*v^3 - 9*v^2 + 31*v - 26
u - v^3 + 3*v - 5
> Type(m);
MapSch
> Inverse(m);
Mapping from: Aff: A to Aff: A
with equations :
-u^2 + 2*u*v^3 - 6*u*v + 10*u - v^6 + 6*v^4 - 10*v^3 - 9*v^2 + 31*v - 26
u - v^3 + 3*v - 5
and inverse
u^3 + 3*u^2*v^2 + 3*u^2 + 3*u*v^4 + 6*u*v^2 + v^6 + 3*v^4 + v + 3
u + v^2 + 1
```

The automorphism test returns two values. The first, `true`, confirms that the map m is an automorphism. In doing this test, Magma computes the inverse and stores it with m. The second is the same map m, now with its inverse computed, but with the type of an automorphism.

```
> _,maut := IsAutomorphism(m);
> maut;
Mapping from: Aff: A to Aff: A
with equations :
u^3 + 3*u^2*v^2 + 3*u^2 + 3*u*v^4 + 6*u*v^2 + v^6 + 3*v^4 + v + 3
u + v^2 + 1
and inverse
-u^2 + 2*u*v^3 - 6*u*v + 10*u - v^6 + 6*v^4 - 10*v^3 - 9*v^2 + 31*v - 26
u - v^3 + 3*v - 5
> Type(maut);
MapAutSch
```

112.14.6.1 Affine Automorphisms

The first constructor below checks that the proposed map is indeed an automorphism by computing an inverse for the map determined by the arguments. It is a potentially expensive test. The other constructors are all either clearly automorphisms or else require only very simple tests.

`Automorphism(A,F)`

The automorphism of the affine space A determined by the sequence of functions F defined on A.

`Automorphism(A,M)`

The linear automorphism of the affine space A determined by the entries of the matrix of base ring elements M acting from the right on points.

`Translation(A,p)`

The translation map of the affine space A taking the rational point p to the origin.

`PermutationAutomorphism(A, g)`

`Automorphism(A, g)`

The automorphism of the affine space A that permutes its coordinates according to the permutation g.

Example H112E41

Permutations are easy to create as elements of the symmetric group. The symmetric group used must act on the set of n points, where n is the dimension of the affine space, even if it is only intended to permute a few of the coordinates.

```
> A := AffineSpace(Rationals(),5);
> g := SymmetricGroup(5) ! (1,2,3);
> f := PermutationAutomorphism(A,g);
> p := A ! [1,2,3,4,5];
> f(p);
(2, 3, 1, 4, 5)
```

`Automorphism(A,p)`

The automorphism which takes the first coordinate x of the affine space A to $x+p$. The polynomial p must be a function on A which does not involve x.

`AffineDecomposition(f)`

If f is an affine linear endomorphism, that is, an automorphism of some affine space defined by polynomials of degree at most 1, this returns a linear endomorphism ℓ and a translation t such that, in Magma notation, `f eq l * t`.

Example H112E42

In this example we make an affine linear map by composing a linear map and a translation. Then we promptly decompose it into these components.

```
> A<x,y,z> := AffineSpace(Rationals(),3);
> f := Automorphism(A,2*y+3*z) * Translation(A,A ! [2,3,5]);
> l,t := AffineDecomposition(f);
> l,t;
Mapping from: Aff: A to Aff: A
with equations :
x + 2*y + 3*z
y
z
and inverse
x - 2*y - 3*z
y
z
Mapping from: Aff: A to Aff: A
with equations :
x - 2
y - 3
z - 5
and inverse
x + 2
y + 3
z + 5
> f eq l * t;
true
```

Note that the composition `l * t` in Magma means that the map ℓ is applied first followed by t.

```
> p := A ! [1,2,3];
> f(p);
(12, -1, -2)
> t(l(p));
(12, -1, -2)
```

`NagataAutomorphism(A)`

This intrinsic returns the Nagata automorphism in a standard form:

$$(u, v, w) \mapsto (-u^2w^3 - 2uv^2w^2 - 2uvw + u - v^4w - 2v^3, uw^2 + v^2w + v, w).$$

Recall that this is an automorphism of affine 3-space A which is not known to be tame, that is, admits no known factorisation into automorphisms of the types listed above.

```
Projectivity(A,M)
```

The restriction to the affine space A of the linear automorphism of its projective closure determined by the matrix M. Note that this map is not usually regular on A but it is an isomorphism where it is defined.

Example H112E43

Most projectivities on an affine space are not regular maps. By definition, the equations which define them are naturally rational polynomials. That is the reason for naming the variables in the field of fractions of the coordinate ring of A in the following code.

```
> k := FiniteField(23);
> A<x,y,z> := AffineSpace(k,3);
> M := Matrix(k,4,4,[1,2,3,-4,2,3,5,6,3,4,5,9,4,5,6,0]);
> phi := Projectivity(A,M);
> KA<u,v,w> := Parent(x/y);
> phi;
Mapping from: Aff: A to Aff: A
with equations :
(6*u + 12*v + 18*w + 22)/(u + 7*v + 13*w)
(12*u + 18*v + 7*w + 13)/(u + 7*v + 13*w)
(18*u + v + 7*w + 8)/(u + 7*v + 13*w)
and inverse
(11*u + 15*v + 5)/(u + 20*w + 2)
(9*u + 16*v + w + 12)/(u + 20*w + 2)
(12*u + 15*v + 3*w + 16)/(u + 20*w + 2)
```

Notice that the inverse of ϕ has also been computed. In fact, ϕ has been returned as an automorphism even though it is not regular.

```
> Type(phi);
MapAutSch
> IsRegular(phi);
false
```

112.14.6.2 Projective Automorphisms

As in the case of affine spaces, a version of the automorphism constructor is provided which takes a sequence of polynomials as second argument. It is included mainly for completeness whereas the other constructors are more appropriate for constructing automorphisms in the contexts in which they often arise.

Projective automorphisms (which are regular) are always linear so they have an associated matrix with respect to the basis of monomials. A function is provided to retrieve this matrix, and conversely automorphisms may be created using matrices. In fact, if the projective space is defined over a finite field, then the automorphism group can be computed (as a group in Magma) and its elements can be realised as matrices.

Also included are functions for creating a nonregular birational automorphism of projective space, the standard *quadratic transformation*. When working in the plane, this together with linear automorphisms generates the group of birational automorphisms. An example of factorising a birational automorphism in this group is given.

```
Automorphism(P,F)
```

The automorphism of the projective space P determined by the sequence of polynomials F defined on P.

```
Matrix(f)
```

The matrix corresponding to the linear automorphism f of a projective space.

```
Automorphism(P,M)
```

The linear automorphism of the projective space P determined by the entries of the matrix of base ring elements M acting on the left of coordinates.

```
Aut(P)
```

The parent of automorphisms of the projective space P.

```
AutomorphismGroup(P)
```

The automorphism group of the projective space P together with a map from this group to the set of automorphisms of P, that is, the parent of such automorphisms. The space P must be defined over a finite field for this intrinsic. Note that currently the group returned is a general linear group rather than the projectivised version. This will be changed in the future, but in any case does not create much confusion.

Example H112E44

When a projective space is defined over a finite field, then its automorphism group can be realised as a group of matrices in a natural way. First we see how to use standard intrinsics to realise the correspondence between matrices and linear automorphisms of projective space.

```
> P<x,y,z> := ProjectiveSpace(GF(5),2);
> phi := Automorphism(P,[x+y,y,z]);
> M := Matrix(phi);
> M;
[1 0 0]
[1 1 0]
[0 0 1]
> Automorphism(P,M) eq phi;
true
```

Since P is a projective space defined over a finite field, we can actually work with a group which is isomorphic to its automorphism group. The map m computed below maps matrices to automorphisms and conversely its inverse constructs a matrix from an automorphism.

```
> G,m := AutomorphismGroup(P);
> G;
```

```
GL(3, GF(5))
> m;
Mapping from: GrpMat: G to Parent structure for automorphisms of P
> phi eq m(Transpose(M));
true
> Transpose(phi @@ m);
[1 0 0]
[1 1 0]
[0 0 1]
```

The parent of automorphisms is also an object in Magma. It can be created using `Aut(P)`.

```
> Aut(P);
Set of all automorphisms of P
> Aut(P) eq Codomain(m);
true
```

`TranslationOfSimplex(P,Q)`

The unique automorphism of the n-dimensional projective space P taking the $n+2$ standard simplex points $(1 : 0 \ldots : 0), \ldots$ and $(1 : 1 \ldots : 1)$ to the points of the sequence Q. The sequence Q must comprise $n + 2$ linearly independent rational points of P.

`Translation(P,Q)`

This function returns an automorphism which translates the standard coordinate points to the points of the sequence Q. The sequence Q must comprise $n+1$ linearly independent rational points of the projective space P where n is the dimension of P. This intrinsic puts no condition on the usual final point $(1 : 1 \ldots : 1)$ of the standard simplex. In other words, it creates the automorphism of P by the matrix having the coordinates of the $n+1$ points of Q as its columns. This automorphism is not uniquely determined since $\mathrm{PGL}(n, k)$ is $n + 2$ transitive.

`Translation(P,p,q)`

A choice of linear automorphism of the projective space P which takes the rational point p to the rational point q.

`Translation(X,p)`

A choice of linear automorphism of the projective space P taking the point $(0 : \ldots : 0 : 1)$ to the rational point p.

Example H112E45

The intrinsic `Translation(X,p)` works in both the affine and the projective context. For an affine scheme, it makes the translation which moves the point p to the origin.

```
> A<u,v> := AffineSpace(Rationals(),2);
> Translation(A,A![1,2]);
Mapping from: Aff: A to Aff: A
with equations :
u - 1
v - 2
and inverse
u + 1
v + 2
```

In the projective case, the resulting translation moves the point p to the image of the origin on the first affine patch. When the point p lies on the first affine patch, then the translation is the obvious one. But when it doesn't a permutation of the coordinates is made first.

```
> P<x,y,z> := ProjectiveSpace(Integers(),2);
> p := P ! [3,2,1];
> f := Translation(P,p);
> f;
Mapping from: Prj: P to Prj: P
with equations :
-x + 3*z
-y + 2*z
z
and inverse
-x + 3*z
-y + 2*z
z
> f(p);
(0 : 0 : 1)
> p := P ! [0,1,0];
> f := Translation(P,p);
> f(p);
(0 : 0 : 1)
> f;
Mapping from: Prj: P to Prj: P
with equations :
-x
-z
y
and inverse
x
-z
y
```

```
QuadraticTransformation(P)
QuadraticTransformation(P,Q)
```

The first function is the standard quadratic transformation of projective space P taking its coordinates to their reciprocals, that is $(x : y : \ldots) \mapsto (1/x : 1/y : \ldots)$. The second conjugates the standard map with a translation of the points of Q to the standard basis vectors. The sequence Q must comprise $n + 1$ linearly independent rational points of P where n is the dimension of P.

```
QuadraticTransformation(X)
QuadraticTransformation(X,Q)
```

The *birational* pullback of the projective scheme X by the quadratic transformation. In the first intrinsic the transformation used is `QuadraticTransformation(P)` while in the second, the transformation is `QuadraticTransformation(P,Q)` where P is the ambient projective space of X. Thus Q must comprise $n+1$ linearly independent rational points of P where n is the dimension of P. Exceptional components in the total pullback of X are removed.

Example H112E46

This example shows how to factorise a simple Cremona transformation. (The reader who knows something about this will note that in this example we take no account of the Nöther–Fano inequalities nor do we analyse infinitely near points. In fact, we are rather lucky to be able to complete the factorisation.)

```
> k := RationalField();
> P<x,y,z> := ProjectiveSpace(k, 2);
> funs := [ 2/3*x^2*y^2 + 2/3*x^2*y*z + x*y^2*z + x*y*z^2,
>     1/3*x^2*y^2 + 4/3*x^2*y*z + x^2*z^2 + 1/2*x*y^2*z + 1/2*x*y*z^2,
>     2/9*x^2*y^2 + 2/3*x^2*y*z + 2/3*x*y^2*z + x*y*z^2 + 1/2*y^2*z^2 ];
> g := map< P -> P | funs >;
> FunctionDegree(g);
4
> RationalPoints(BaseScheme(g));
{@ (3/4 : -1 : 1), (0 : 1 : 0), (0 : 0 : 1), (-3/2 : -1 : 1), (1 : 0 : 0) @}
```

Our aim is to precompose g with quadratic transformations which will reduce the degree of its defining polynomials, currently 4. When the function degree is reduced to 1 the factorisation is complete since the inverse sequence of quadratic transformations will comprise a factorisation of g up to a translation. (Draw the diagram of maps!) That will be done at the end. First a quadratic transformation in the three coordinate points is made since these points are the simplest appearing in the list of base points. (The complete mathematical theory works hard to make the choice of map here the right one: clearly no hard work has been done here in making the choice of coordinate points.)

```
> std_quad := QuadraticTransformation(P);
> g1 := std_quad * g;
> // (Expand and) extend the map to its maximal domain:
```

```
> g1:=Extend(g1);
> FunctionDegree(g1);
2
> S := BaseScheme(g1);
> RationalPoints(S);
{@ (4/3 : -1 : 1), (-2/3 : -1 : 1), (-2/3 : 0 : 1) @}
> HasPointsOverExtension(S);
false
```

Good! With only three non-collinear points of indeterminacy (the final line makes sure there aren't further points defined over some extension of the base ring) we are only one step away from completing the factorisation. We make a quadratic transformation in these three noncollinear points.

```
> tr := Translation(P,[ p : p in $2 ]);
> quad := std_quad * tr;
> g2 := quad * g1;
> g2:=Extend(g2);
> FunctionDegree(g2);
1
```

So g is seen to be the composition of linear translations and standard quadratic transformations: g_2 is itself a linear translation. This sequence of maps is reconstructed in reverse order to get g. All the maps should be inverted, but note that the quadratic transformations are selfinverse so that this is rather easy. Of course, composing the maps in the order they were discovered above produces the birational inverse of g.

```
> f3 := f2 * g2
>     where f2 is f1 * std_quad
>     where f1 is std_quad * Inverse(tr);
> Expand(f3) eq g;
true
```

112.14.7 Scheme Graph Maps

In Magma V2.16, we have introduced an alternative to the usual MapSch for maps between *ordinary projective* schemes: a new type MapSchGrph, which we refer to as graph maps. These are objects whose defining data is the closure of the graph of a rational map, without explicit defining or inverse polynomials. This is a fairly traditional way to consider maps in algebraic geometry and the main motivation for their introduction is for divisor maps of invertible sheaves. These are naturally constructed in graph form and the further derivation of explicit defining polynomials can be quite time-consuming and lead to extremely high degree, ugly results.

As well as being more naturally constructible in certain situations, graph maps have advantages over MapSchs in a number of contexts.

1 A graph maps is automatically maximally defined, so Extend and alternative equations are unnecessary.

2 Computation of images of subschemes of the domain or of the inverse of a map go, in one way or another, through the graph of the map, so it is more efficient to already have it in graph form.

3 For an invertible graph map, separate inverse equations are not required. It is only necessary to record that it is invertible and consider the "reverse" of the graph.

If $f : X \to Y$ is a rational map, the closure of its graph G naturally lies in $X \times Y$. For computational ease, we take G as lying in the product projective space of the ambients of X and Y, Functionally, it is defined by a bihomogenous ideal in a polynomial ring with $n + m + 2$ variables, where n (resp. m) is the dimension of the ambient of X (resp. Y). There is a primitive basic user constructor described below. Graph maps are more naturally constructed and returned by specialised functions.

Graph maps have most of the functionality of `MapSch` maps including `IsInvertible` and `Expand`. Rather than repeat the list of all of the relevant intrinsics, we note here that the major functionality not currently available for them is

1 No defining or inverse defining polynomials.

2 No pointset map construction: it is not possible to ask for the image or preimage under a graph map of a point in a pointset over a proper extension of the base field. Neither are there images and preimages of rational functions.

3 Restriction to affine patchs of the domain or codomain is not possible (graph maps are only between projective schemes). However, restriction to a closed subscheme of the domain or codomain via `Restriction` is allowable as usual.

4 Any specialised `MapSch` functionality only available for maps between particular scheme types, like maps between curves.

Graph maps can be composed in the usual way for maps, but can not be mixed with `MapSch` maps in composition.

A further minor restriction has been built in for implementational efficiency. It is assumed for the domain X of a graph map that there is no coordinate hyperplane that contains some *but not all* of the irreducible components of X. In particular, there is no problem if X is irreducible.

A graph map f may be converted into normal `MapSch` with an intrinsic given below. If f is known invertible, this also computes inverse defining polynomials. It should be noted that for maps between complicated schemes, this often produces a `MapSch` with extremely high degree defining polynomials and a large base scheme where it is not defined. In such cases, the original `MapSchGrph` can be a functionally much more efficient representation.

```
SchemeGraphMap(X, Y, I)
```

`Saturated` BoolElt *Default :* `false`

X and Y are ordinary projective schemes with ambients $\mathbf{P}^m$ and $\mathbf{P}^n$ respectively. I is an ideal in an $n + m + 2$ variable polynomial ring R that should have the grevlex ordering. I should define the closure of the graph of a rational map from X to Y if R is identifed with the coordinate ring of the product projective space of $\mathbf{P}^m$ and

$\mathbf{P}^n$ such that the first $m+1$ variables correspond to the variables of the coordinate ring of $\mathbf{P}^m$ in the same order and the last $n+1$ variables correspond to those of $\mathbf{P}^n$ in the same order.

I should thus be bihomogeneous in the first $m+1$ and second $n+1$ variables and be large enough to define the graph of the map, though it doesn't have to be the maximal defining ideal. For example, if a map is explicitly given by defining polynomials $[F_0(x), \ldots, F_n(x)]$ where we use x_i and y_i to denote the variables of R corresponding to the domain and codomain variables, then the ideal generated by

$$y_i F_j(x) - y_j F_i(x) \quad \forall\ 0 \le i, j \le n$$

and the defining equations for X will define the graph if the defining polynomials give an everywhere defined map. We give an explicit example below. If there is a non-empty base scheme, it will be necessary to saturate the above ideal by one of the $F_i(x)$ that doesn't vanish on any component of the domain. It is then "domain" saturated.

A defining ideal like the above that *hasn't* been saturated by an $F_i(x)$ is usually not maximal and for functional purposes has to be saturated by an appropriate domain variable using `ColonIdeal`. This is performed internally. If the user already knows that I is domain saturated, he can set the parameter `Saturated` (default `false`) to `true` to avoid this.

This is a simple convenience function that performs practically no checks on the validity of the input data.

`SchemeGraphMapToSchemeMap(f)`

Converts the graph map f into a usual scheme map. As noted in the introduction, if f is a map between fairly complex schemes, this can be quite a computationally heavy procedure and can produce very large degree, non-sparse defining polynomials and the `MapSch` produced can have a large base scheme, even if f is defined everywhere on its domain. If f is known invertible (see below), inverse defining-polynomials are also added to the result.

`IsInvertible(f)`

As for the `MapSch` version, returns whether f is birationally invertible and, if so, also returns the inverse map. This records that f is known invertible internally, as the inverse map just has the graph reversed (also the graph ideal may need to be saturated with respect to a codomain variable, which will be performed here if it is determined that the map is invertible) so no inverse equations are added. The `HasKnownInverse` intrinsic for `MapSchs`, which returns whether the map has already been determined to be invertible, is also available.

Example H112E47

Let X be an elliptic curve, given as an intersection of quadrics in $\mathbf{P}^3$. Here we start with a map from X into $\mathbf{P}^4$, given by cubics, that is birational onto its image. We show how to convert this into a graph map, take restrictions, check for invertibility and various other things.

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Scheme(P3,[x*t-y*z, x^2+y^2-4*z^2+7*t^2]);
> P4<a,b,c,d,e> := ProjectiveSpace(Rationals(),4);
> mp_seq := [x^3,y^3,z^3,t^3,y*z*t]; // polys defining a map to P^4
> mp := map<X->P4|mp_seq>;
> // Will now define the graph map
> R<x1,x2,x3,x4,y1,y2,y3,y4,y5> := PolynomialRing(Rationals(),9,"grevlex");
> hm := hom<CoordinateRing(P3) -> R |[R.i : i in [1..4]]>; // usual map
> grI := ideal<R|[(R.(i+4))*hm(mp_seq[j])-(R.(j+4))*hm(mp_seq[i]):
>   i in [j+1..5] , j in [1..5]] cat [hm(b) : b in Basis(Ideal(X))]>;
> gr_mp := SchemeGraphMap(X,P4,grI); // the graph map
> // check that gr_mp does give mp
> mp eq SchemeGraphMapToSchemeMap(gr_mp);
true
> Y := gr_mp(X);
> Y eq mp(X);
true
> // take restrictions to the image
> gr_mp1 := Restriction(gr_mp,X,Y);
> mp1 := Restriction(mp,X,Y);
> // check that gr_mp1 still gives mp1
> mp1 eq SchemeGraphMapToSchemeMap(gr_mp1);
true
> boo := IsInvertible(gr_mp1);
> boo;
true
> // find the image and preimage of a point
> pt := X![2,0,1,0];
> ipt := gr_mp1(X![2,0,1,0]);
> iminv := ipt @@ gr_mp1;
> Dimension(iminv); Degree(iminv);
0
2
> // the preimage is just pt doubled
> Support(iminv);
{ (2 : 0 : 1 : 0) }
```

112.15 Tangent and Secant Varieties and Isomorphic Projections

The functions in this section relate to the isomorphic projection of schemes in higher-dimensional projective spaces down to lower-dimensional ones. This is achieved through finding points in the ambient projective space which don't lie on either the tangent or secant varieties of the scheme. These varieties are interesting in their own right and we provide functions to compute them as subschemes of the ambient space or to test if a given point lies on them.

112.15.1 Tangent Varieties

For a scheme X in affine or ordinary projective space over a field, the tangent variety TX is a subscheme of the ambient space whose set of closed points is the closure of the union of all tangent spaces of closed points of X. We do not worry if the TX that we construct is necessarily a reduced scheme or not. If X is non-reduced, TX probably won't be and will usually be of larger dimension than expected. If X is projective, the union of tangent spaces is already closed in the ambient space.

`TangentVariety(X)`

`PatchIndex` RngIntElt *Default :* 0

The scheme X must be affine or ordinary projective. If X is projective, the tangent variety can be computed projectively but it is usually quicker to compute the result for an affine patch and then take the projective closure. If the parameter `PatchIndex` is set to $i > 0$ then in the projective case the function will do this, using the ith standard affine patch (see `AffinePatch` on page 3517). This will give the correct result as long as *no component of X lies in the hyperplane complement of the patch.*

`IsInTangentVariety(X,P)`

The computation of the full tangent variety can be quite time consuming except in small dimensional ambient spaces. If the dimension of the ambient space is n, it is effectively calculated as the image of a subscheme in a $2n$-dimensional ambient under projection down to X's ambient space. However, this call gives a much faster way of testing if a particular point P in the ambient space lies in the tangent variety of the scheme X when X is projective.

Example H112E48

```
> P<x,y,z,t> := ProjectiveSpace(RationalField(),3);
> X := Scheme(P,[x*y+z*t,x^2-y^2+2*z^2-4*t^2]);
> Dimension(X);
1
> time TangentVariety(X);
Scheme over Rational Field defined by
x^8 + 4*x^6*y^2 + 25/4*x^6*z^2 - 25/2*x^6*t^2 + 44*x^5*y*z*t + 6*x^4*y^4 +
    25/4*x^4*y^2*z^2 - 25/2*x^4*y^2*t^2 + 27/2*x^4*z^4 - 193/8*x^4*z^2*t^2 +
    54*x^4*t^4 + 88*x^3*y^3*z*t + 275/2*x^3*y*z^3*t - 275*x^3*y*z*t^3 +
    4*x^2*y^6 - 25/4*x^2*y^4*z^2 + 25/2*x^2*y^4*t^2 - 67/2*x^2*y^2*z^4 +
    2025/4*x^2*y^2*z^2*t^2 - 134*x^2*y^2*t^4 + 11*x^2*z^6 + 22*x^2*z^4*t^2 -
    44*x^2*z^2*t^4 - 88*x^2*t^6 + 44*x*y^5*z*t - 275/2*x*y^3*z^3*t +
    275*x*y^3*z*t^3 + 50*x*y*z^5*t + 200*x*y*z^3*t^3 + 200*x*y*z*t^5 + y^8 -
    25/4*y^6*z^2 + 25/2*y^6*t^2 + 27/2*y^4*z^4 - 193/8*y^4*z^2*t^2 + 54*y^4*t^4
    - 11*y^2*z^6 - 22*y^2*z^4*t^2 + 44*y^2*z^2*t^4 + 88*y^2*t^6 + 2*z^8 +
    16*z^6*t^2 + 48*z^4*t^4 + 64*z^2*t^6 + 32*t^8
Time: 0.440
> time TangentVariety(X: PatchIndex := 4);
Scheme over Rational Field defined by
x^8 + 4*x^6*y^2 + 25/4*x^6*z^2 - 25/2*x^6*t^2 + 44*x^5*y*z*t + 6*x^4*y^4 +
    25/4*x^4*y^2*z^2 - 25/2*x^4*y^2*t^2 + 27/2*x^4*z^4 - 193/8*x^4*z^2*t^2 +
    54*x^4*t^4 + 88*x^3*y^3*z*t + 275/2*x^3*y*z^3*t - 275*x^3*y*z*t^3 +
    4*x^2*y^6 - 25/4*x^2*y^4*z^2 + 25/2*x^2*y^4*t^2 - 67/2*x^2*y^2*z^4 +
    2025/4*x^2*y^2*z^2*t^2 - 134*x^2*y^2*t^4 + 11*x^2*z^6 + 22*x^2*z^4*t^2 -
    44*x^2*z^2*t^4 - 88*x^2*t^6 + 44*x*y^5*z*t - 275/2*x*y^3*z^3*t +
    275*x*y^3*z*t^3 + 50*x*y*z^5*t + 200*x*y*z^3*t^3 + 200*x*y*z*t^5 + y^8 -
    25/4*y^6*z^2 + 25/2*y^6*t^2 + 27/2*y^4*z^4 - 193/8*y^4*z^2*t^2 + 54*y^4*t^4
    - 11*y^2*z^6 - 22*y^2*z^4*t^2 + 44*y^2*z^2*t^4 + 88*y^2*t^6 + 2*z^8 +
    16*z^6*t^2 + 48*z^4*t^4 + 64*z^2*t^6 + 32*t^8
Time: 0.040
> time IsInTangentVariety(X,P![1,2,3,4]);
false
Time: 0.000
```

112.15.2 Secant Varieties

For a scheme X in affine or ordinary projective space over a field, the secant variety SX is a subscheme of the ambient space whose set of closed points is the closure of the union of all lines joining distinct pairs of closed points of X (secants). Again, we do not worry if the SX that we construct is necessarily a reduced scheme or not. Note that the union of all secants is not necessarily a closed subset of the ambient space even when X is projective.

```
SecantVariety(X)
```

`PatchIndex` RngIntElt *Default :* 0

The scheme X must be affine or ordinary projective. In the projective case we construct SX by taking the projective closure of the result for an appropriate affine patch (intersecting every component of X). For simplicity, we currently only consider standard affine patchs so the function will fail for X projective if *it has components lying in every standard hyperplane.* As for `TangentVariety`, if the parameter `PatchIndex` is set to $i > 0$ then the ith standard affine patch will be the one used and this saves a little time, avoiding a search. Effectively, the computation consists of finding the image of a subscheme in a projection from a $2n + 1$ dimensional ambient space down to the (n dimensional) ambient space of X and can be quite lengthy.

```
IsInSecantVariety(X,P)
```

Again as in the tangent variety case, if the scheme X is projective, this call gives a much faster way of testing if a given ambient point P lies in SX than computing the whole of SX with `SecantVariety`. To be precise, this call actually tells you whether or not P lies in the union of secants rather than its closure SX. Additionally, affine patchs are not used, so the above restriction on validity doesn't apply.

Example H112E49

```
> P<a,b,c,d,e> := ProjectiveSpace(RationalField(),4);
> X := Scheme(P,[a*d + c*e, a*c + d*e,
>  a^2 - e^2, c^2*e - d^2*e,
>  b^2 + c*d + e^2]); // union of 3 irreducible curves
> Dimension(X);
1
> time SecantVariety(X : PatchIndex := 2);
Scheme over Rational Field defined by
a^4*b^2 + a^4*e^2 - a^3*c^2*e - a^3*d^2*e + a^2*b^4 + 2*a^2*b^2*c*d +
    1/4*a^2*c^4 + 1/2*a^2*c^2*d^2 + 1/4*a^2*d^4 - a^2*e^4 + a*c^2*e^3 +
    a*d^2*e^3 - b^4*e^2 - 2*b^2*c*d*e^2 - b^2*e^4 - 1/4*c^4*e^2 -
    1/2*c^2*d^2*e^2 - 1/4*d^4*e^2
Time: 26.890
> Dimension($1);
3
> time IsInSecantVariety(X,P![0,1,0,-3,0]);
true
Time: 0.000
```

112.15.3 Isomorphic Projection to Subspaces

The aim of the functions here is to try to find embeddings of projective schemes which lie in high-dimensional ambient spaces into lower dimensional spaces via projection.

Let X be a scheme in ordinary projective space that is assumed to be *reduced* but not necessarily irreducible or non-singular, d the dimension of X and n the dimension of its ambient space P. The conditions imply that the tangent variety and secant variety of X are finite unions of subschemes of dimension at most $2d + 1$ (*cf* [Har77] IV.3), so while $n > 2d + 1$, we can find points in P lying in neither of these, unless possibly the base field is a finite field. For such a point, projection down to a hyperplane gives an isomorphism of X to its image (*cf. op. cit.*) and we can continue projecting down one dimension at a time until $n = 2d + 1$.

In fact, the tangent and secant varieties of the image of X will be their images under the projection also, which induces a finite map on them (it's quasi-finite since the inverse image of a point in the image hyperplane is a line through the projecting point *pt*, which any (closed) subscheme not containing *pt* must intersect finitely). Hence their images have the same dimensions. If the maximum of these dimensions is $tsd \leq 2d+1$ we can actually project down to a subspace of dimension tsd. Further, the fact that the tangent and secant images correspond under projection shows that if we find a linear change of variables for the coordinates of P, $(x_1,\ldots,x_n) \to (y_1,\ldots,y_n)$, such that $y_1,\ldots,y_{tsd}$ are a set of Noether normalising variables for the defining ideal of the union of the tangent and secant varieties in the coordinate ring of P, then we can project X isomorphically down to the tsd dimensional linear subspace $y_{tsd+1}=0,\ldots,y_n=0$ directly.

This would be the most elegant solution. However, in practise it involves computing the full tangent and secant varieties which can be extremely time-consuming once we are in dimensions above around 3 or 4. By contrast, checking if a random point of P lies in the secant or tangent variety is reasonably fast. Therefore, we choose to follow the method of picking random points to project down one dimension at a time and finish when we reach dimension $2d+1$ (when the secant variety generally fills the ambient space anyway).

```
IsomorphicProjectionToSubspace(X)
```

Verbose `IsoToSub` *Maximum* : 1

As described above, this function projects the scheme X isomorphically down to a linear subspace of its ambient space P of dimension $2d+1$, if $2d+1 < n$. In the case of finite base fields or infinite ones of small characteristic, the process may stop before reaching this dimension but this should be very rare (difficulties finding random points outside the secant/tangent varieties). The return values consist of the image scheme, with the subspace as its ambient space, and the explicit map taking X to this image.

```
EmbedPlaneCurveInP3(C)
```

Verbose `EmbCrv` *Maximum* : 1

This function embeds a plane curve C as a non-singular projective curve in ordinary 2- or 3-dimensional projective space over the base field. This is effected by first using the function field machinery to give a non-singular projective embedding and then projecting down to a 3-dimensional subspace if necessary. In rare cases with small finite base fields or in small characteristic, the final embedding may still be into a higher-dimensional ambient space. The image scheme is returned along with the mapping of C to it.

Example H112E50

We will embed a genus 5 plane curve into $P3$.

```
> P<x,y,z> := ProjectiveSpace(RationalField(),2);
> f := x^5+x^2*y^3+y^2*z^3+x^2*z^3-y*z^4-z^5;
> C := Curve(P,f);
> Genus(C);
5
```

```
> SetVerbose("IsoToSub",1);
> SetVerbose("EmbCrv",1);
> C1, mp := EmbedPlaneCurveInP3(C);
Curve of genus 5.
Mapping to 4-space by canonical divisor map
Map was an embedding (non-hyperelliptic curve).
Beginning projection to 3-space.
Projection from dimension 4 to dimension 3:
Finding good projection point...
Performing projection...
Time: 0.002
> P1 := AmbientSpace(C1);
> AssignNames(~P1,["a","b","c","d"]);
> C1;
Scheme over Rational Field defined by
a^2*d^3 + a*b^2*c*d - a*b*c^2*d + a*b*d^3 + a*c^2*d^2 - a*d^4 + b^5 + b^2*c^3 -
    b^2*c*d^2 - b^2*d^3 + 2*b*d^4 - c^2*d^3 - d^5,
a^2*c*d + a*b^3 + a*c^3 - a*c*d^2 + b*c*d^2 - c*d^3,
a*b^2*d - a*b*d^2 + a*c^2*d + b^3*c + c^4 - c^2*d^2,
a^3*d + a^2*c^2 - a^2*d^2 + a*b*d^2 - a*d^3 + b^2*c*d,
a^2*b - c^2*d
> Dimension(C1);
1
> ArithmeticGenus(C1);
5
> IsNonSingular(C1);
true
```

112.16 Linear Systems

Let $f_1, \ldots, f_r$ be homogeneous polynomials of some common degree d on some projective space $\mathbf{P}$ defined over a field. The set of hypersurfaces

$$a_1 f_1 + \ldots + a_r f_r = 0$$

where the a_is are elements of the base field of $\mathbf{P}$ is an example of a linear system. This can be thought of as being the vector space of elements $(a_1, \ldots, a_r)$ or even the projectivisation of that space (since multiplying the equation above by a constant doesn't change the hypersurface it defines and the equation $0 = 0$ doesn't define a hypersurface at all). The same is true if $f_1, \ldots, f_r$ are a finite collection of polynomials defined on some affine space.

All linear systems in Magma arise in a similar way to the above example. It doesn't matter whether the linear system is considered to be the collection of hypersurfaces or the collection of homogeneous polynomials or the vector space of coefficients; Magma allows each of these interpretations and the distinction is blurred in the text below. One should note that linear systems in Magma are being used in a very elementary way: compare with the discussion on plane conics and cubics in the first two chapters of Reid's Student Text [Rei88].

Immediate applications of linear systems arise because of their close relationship to maps (consider the map to an $r-1$-dimensional projective space defined by the polynomials $f_1, \ldots, f_r$) and their application to the extrapolation a scheme of some particular degree from a set of points lying on it or some subscheme of it.

More ambitious interpretations, as the zero-th coherent cohomology group of an invertible sheaf for instance, cannot be realised explicitly in Magma except inasmuch as the user can understand input and output easily in these terms. There is no analysis of linear systems on general schemes and so, in particular, no analysis of exact sequences of cohomology groups.

We give a brief description of the way in which linear systems work in Magma, an approach which echoes the more general definition. The *complete linear system on* $\mathbf{P}$ *of degree* d is the collection of all homogeneous polynomials of degree d on $\mathbf{P}$, or equivalently, the degree d hypersurfaces they define. Magma does not consider this to be a unique object: each time such a system is created, a completely new object will be created distinct from any previous creation. Its major attributes include a particular basis of degree d polynomials, which is always the standard monomial basis, and a vector space whose vectors correspond to the coefficients of a polynomial with respect to this basis. The vector space is called the *coefficient space* of the linear system. There are comparison maps: one to produce the vector of coefficients of polynomials with respect to the given basis and one to create a polynomial from a vector of coefficients. Most questions involving the analysis of linear systems are translated into the linear algebra setting, solved there and then translated back.

A general linear system corresponds to some vector subspace of the coefficient space of a complete linear system. The correspondence between vectors of coefficients and polynomials are computed at the level of the complete systems so that any two subsystems interpret coefficient vectors with respect to the same basis of polynomials.

112.16.1 Creation of Linear Systems

In practice, linear systems are not often created explicitly by hand. Typically, the complete linear system of all hypersurfaces of a given degree is created and then restricted by the imposition of geometrical conditions. For example, such a condition could require that all hypersurfaces pass through a particular point.

The creation methods below are split into three classes: (i) explicit initial creation methods; (ii) methods of imposing geometrical conditions; and (iii) creation of subsystems by nominating specific technical data calculated in advance by the user.

112.16.1.1 Explicit Creation

Initially, we present three methods by which a linear system can be created. The complete linear system of degree d whose sections are all monomials of that degree has a special creation function. Alternatively, a sequence of monomials of some common degree can be specified to generate the sections of a linear system. The third constructor is useful for calculating the images of maps and has been seen before: given a scheme S and a map f it calculates the linear system of hypersurfaces which contain $f(S)$.

`LinearSystem(P,d)`

The complete linear system on the affine or projective space P of degree d. In the projective case, this is the space of all homogeneous polynomials of degree d on P, whereas in the affine case it includes all polynomials of degree no bigger than d. The integer d must be strictly positive.

`LinearSystem(P, d)`

The complete linear system on the affine or projective space P of multi-degree d. The length of d must be the number of gradings of P, i.e. one degree for each grading. In the projective case, this is the space of all homogeneous polynomials of degree d on P, whereas in the affine case it includes all polynomials of degree no bigger than d. The integers in d must be strictly positive.

`LinearSystem(P,F)`

If P is a projective space and F is a sequence of homogeneous polynomials all of the same degree defined on P, or if P is an affine space and F is a sequence of polynomials defined on P this returns the linear system generated by these polynomials. If the polynomials in F are linearly independent they will be used as a basis of the sections of the resulting linear system, otherwise a new basis will be computed.

`MonomialsOfWeightedDegree(X, D)`

Return the monomials in the coordinate ring of the ambient of X having degree $D[i]$ with respect to the ith grading of the ambient of X.

Example H112E51

In this example we construct two linear systems on a projective plane. Although they are created in slightly different ways, Magma recognises that they are the same. It does the computation as a subspace equality test in the corresponding 'coefficient spaces'.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> L := LinearSystem(P,1);
> K := LinearSystem(P,[x+y,x-y,z+2*z+3*y]);
> L eq K;
true
```

`ImageSystem(f,S,d)`

The linear system on the codomain of the map of schemes f consisting of degree d hypersurfaces which contain $f(S)$. An error is reported if the scheme S does not lie in the domain of f.

Example H112E52

This example demonstrates how one can use an intrinsic based on linear systems, `ImageSystem`, to find the equations of images of maps. The point is that sometimes the usual Gröbner basis can be very difficult, so if one is interested in the equations of low degree then the linear algebra computation might be more convenient.
The curve C has one singularity analytically equivalent to the cusp $u^2 = v^5$ so is well-known to have genus 4.

```
> Q := RationalField();
> P<x,y,z> := ProjectiveSpace(Q,2);
> C := Curve(P,x^5 + y^4*z + y^2*z^3);
```

The canonical embedding of C is therefore given by four conics having a common tangent with the curve at its singularity.

```
> P3<a,b,c,d> := ProjectiveSpace(Q,3);
> phi := map< P -> P3 | [x^2,x*y,y^2,y*z] >;
```

Unless C is hyperelliptic, its canonical image will be the complete intersection of a conic and a cubic in $\mathbf{P}^3$.

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

In this case the Gröbner basis of X has six elements so it is not so helpful for human comprehension. (Compare this with [Hartshorne, IV, Example 5.2.2].)

112.16.1.2 Geometrical Restrictions

Consider the following example. Suppose that L is a linear system on the projective plane whose sections are generated by the monomials x^2, xy, yz and let $p = (1 : 0 : 0)$. The phrase '*one imposes the condition on sections of L that they pass through the point p*' refers to the construction of the subsystem of L, all of whose hypersurfaces pass through p. Explicitly, this involves solving the linear equation in a,b,c obtained by evaluating the equation

$$ax^2 + bxy + cyz = 0$$

at the point p. In this example, the equation is $a = 0$ and the required subsystem is the one whose sections are generated by xy and yz.

The functions described in this section all determine a linear subsystem of a given linear system by imposing conditions on the sections of that system.

```
LinearSystem(L,p)
```

```
LinearSystem(L,S)
```

Given a point p or a sequence S of points, create the subsystem of the linear system L comprising those hypersurfaces of L which pass through p or the points of S.

```
LinearSystem(L,p,m)
```

Create the subsystem of the linear system L comprising hypersurfaces which pass through the point p with multiplicity at least m.

Example H112E53

In this example we make some subsystems of linear systems by imposing conditions at points. In the first example, we construct the family of all curves having singularities with prescribed multiplicities at prescribed points. See Chapter 114 for functions which apply to curves.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> L := LinearSystem(P,6);
> p1 := P ! [1,0,0];
> p2 := P ! [0,1,0];
> p3 := P ! [0,0,1];
> p4 := P ! [1,1,1];
> L1 := LinearSystem(L,p1,3);
> L2 := LinearSystem(L1,p2,3);
> L3 := LinearSystem(L2,p3,3);
> L4 := LinearSystem(L3,p4,2);
> #Sections(L4);
7
> C := Curve(P,&+[ Random([1,2,3])*s : s in Sections(L4) ]);
> IsIrreducible(C);
true
> Genus(C);
0
```

In other words, $L4$ parametrises a six-dimensional family of rational plane curves. (At least, the general element of $L4$ is the equation of a rational plane curve — there are certainly degenerate sections which factorise so don't define an irreducible curve at all.) It would be nice to be able to parametrise one of these curves. The problem is that we need to choose a general one in order that it be irreducible, but on the other hand we have very little chance of finding a rational point of a general curve. However, since this family is nice and big, we can simply impose another point condition on it yielding a rational point on each of the restricted elements.

```
> p5 := P ! [2,1,1];
> L5 := LinearSystem(L4,P![2,1,1]);
> C := Curve(P,&+[ Random([1,2,3])*s : s in Sections(L5) ]);
> IsIrreducible(C);
true
> Genus(C);
0
```

```
> L<u,v> := ProjectiveSpace(Rationals(),1);
> phi := Parametrization(C, Place(C!p5), Curve(L));
> Ideal(Image(phi)) eq Ideal(C);
true
```

To illustrate another feature of imposing point conditions on linear systems, we use a point that is not in the base field of the ambient space. Linear systems on an ambient spaces are defined over its base field, so nonrational points impose conditions as the union of their Galois conjugates.

```
> A<x,y> := AffineSpace(FiniteField(2),2);
> L := LinearSystem(A,2);
> L;
Linear system on Affine Space of dimension 2 Variables : x, y
with 6 sections: 1 x y x^2 x*y y^2
> k1<w> := ext< BaseRing(A) | 2> ;
> p := A(k1) ! [1,w];
> p;
(1, w)
> LinearSystem(L,p);
Linear system on Affine Space of dimension 2
Variables : x, y
with 4 sections:
x^2 + 1
x*y + y
x + 1
y^2 + y + 1
> k2<v> := ext< BaseRing(A) | 3> ;
> q := A(k2) ! [1,v];
> LinearSystem(L,q);
Linear system on Affine Space of dimension 2
Variables : x, y
with 3 sections:
x^2 + 1
x*y + y
x + 1
```

Note the minimal polynomial of the y coordinate of the point $(1, w)$ is of degree 2 so is visible in the restricted linear system. On the other hand, v is of order 3 so it imposes more conditions on the linear system.

LinearSystem(L,X)

The subsystem of the linear system L comprising elements of L which contain the scheme X. The sections of this linear system is equal to the polynomials of the defining ideal of X whose homogeneous degree is the same as that of L.

`LinearSystemTrace(L,X)`

The trace of the linear system L on the scheme X which lies in the ambient space of L. This merely simulates the restriction of the linear system L on X by taking the sections of L modulo the equations of X. The result is still a linear system on the common ambient space.

Example H112E54

In this example, we restrict the linear system of cubics in space to a scheme, which is in fact a twisted cubic curve. (The intrinsic `Sections` is defined in Section 112.16.2.2.)

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> L := LinearSystem(P,3);
> X := Scheme(P,[x*z-y^2,x*t-y*z,y*t-z^2]);
> #Sections(L);
20
> L1 := LinearSystemTrace(L,X);
> #Sections(L1);
10
```

Taking the sections of L modulo the equations of X reduces the dimension of the space of sections by 10. Of course, there is a choice being made about which particular trace sections to use — in the end, the computation is that of taking a complement in a vector space of some vector subspace.

Since we recognise this scheme X as the image of a projective line, we can confirm that the result is correct. We make a map from the projective line to the space P which has image X. Then we check that the pullback L and $L1$ are equal on the projective line. In fact, since the linear system embedding the line is the complete system of degree 3, and L comprises degree 3 hypersurfaces on P, both pullbacks should give the complete linear system on the line of degree $3 \times 3 = 9$. (The intrinsic `Pullback` is defined in Section 112.16.3.)

```
> P1<u,v> := ProjectiveSpace(BaseRing(P),1);
> phi := map< P1 -> P | [u^3,u^2*v,u*v^2,v^3] >;
> Ideal(phi(P1)) eq Ideal(X);
true
> Pullback(phi,L) eq Pullback(phi,L1);
true
> Pullback(phi,L1);
Linear system on Projective Space of dimension 1
Variables : u, v
with 10 sections:
u^9 u^8*v u^7*v^2 u^6*v^3 u^5*v^4 u^4*v^5 u^3*v^6 u^2*v^7 u*v^8 v^9
```

112.16.1.3 Explicit Restrictions

```
LinearSystem(L,F)
```

The subsystem of the linear system L generated by the polynomials in the sequence F. An error results if the polynomials of F are not already sections of L. As before, the polynomials in F are used as a basis of the sections of the resulting linear system provided they are linearly independent, otherwise a new basis is computed.

```
LinearSystem(L,V)
```

The subsystem of the linear system L determined by the subspace V of the complete coefficient space of L. It is an error to call this if V is not a subspace of the coefficient space of L.

112.16.2 Basic Algebra of Linear Systems

This section presents functions for the following tasks: (i) assessing properties of the data type; (ii) geometrical properties of linear systems; (iii) linear algebra operations.

112.16.2.1 Tests for Linear Systems

```
Ambient(L)
AmbientSpace(L)
```

The projective space on which the linear system L is defined.

```
L eq K
```

Returns `true` if and only if the linear systems L and K are equal if considered as linear subsystems of some complete linear system. An error results if L and K lie in different complete linear systems.

```
IsComplete(L)
```

Returns `true` if and only if the linear system L is the complete linear system of polynomials of some degree.

```
IsBasePointFree(L)
IsFree(L)
```

Returns `true` if and only if the linear system L has no base points.

112.16.2.2 Geometrical Properties

`Sections(L)`

A sequence whose elements form basis of the sections of the linear system L. By definition, this is a maximal set of linearly independent polynomials which are elements of L.

`Random(LS)`

If the base field of LS admits random elements then this returns a random element of the space of sections in the linear system LS. If the base field is the rational field, then a section having small random rational coefficients is defined. Otherwise, if there is no random element generator for the base field, the zero section is returned.

`Degree(L)`

The degree of the sections of the linear system L.

`Dimension(L)`

The projective dimension of the linear system L. This is the maximal number of linearly independent sections of L minus 1.

`BaseScheme(L)`

The base scheme of the linear system L. This is simply the scheme defined by the sections of L. This function does not perform any tests on this scheme; it might be empty for example.

`BaseComponent(L)`

The hypersurface common to all the elements of the linear system L.

`Reduction(L)`

The linear system L with its codimension 1 base locus removed. In other words, the linear system defined by the sections of L after common factors are removed.

Example H112E55

If one tries to impose too many point conditions on a linear system, the general elements will no longer be irreducible. From a quick genus calculation one might think that it was possible to impose singularities on multiplicities 2, 3, 4 on projective curves of degree 6 to reveal rational curves — indeed $g = 10 - 1 - 3 - 6 = 0$ if the resulting curves are irreducible.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> L := LinearSystem(P,6);
> p1 := P ! [1,0,0];
> p2 := P ! [0,1,0];
> p3 := P ! [0,0,1];
> L1 := LinearSystem(L,p1,4);
> L2 := LinearSystem(L1,p2,3);
> L3 := LinearSystem(L2,p3,2);
> Sections(L3);
[ x^2*y^3*z, x^2*y^2*z^2, x^2*y*z^3, x^2*z^4, x*y^3*z^2,
    x*y^2*z^3, x*y*z^4, y^3*z^3, y^2*z^4 ]
> BaseComponent(L3);
Scheme over Rational Field defined by z
```

But notice that every section is divisible by z. So the curve $z = 0$ is in the base locus of this linear system, that is, it is contained in every curve in the linear system $L3$. The intrinsic `BaseComponent` identifies this component. The intrinsic `Reduction` creates a new linear system by removing this codimension 1 base locus, as is seen below. First, however, we look at the complete set of prime components of the base scheme and see that, while there is only one codimension 1 component which we already know, there is another component in higher codimension. When we reduce to remove the base component, this other piece of the base scheme remains, and other codimension 2 components also appear.

```
> MinimalPrimeComponents(BaseScheme(L3));
[
    Scheme over Rational Field defined by
    z,
    Scheme over Rational Field defined by
    x
    y
]
> L4 := Reduction(L3);
> Sections(L4);
[ x^2*y^3, x^2*y^2*z, x^2*y*z^2, x^2*z^3, x*y^3*z, x*y^2*z^2,
    x*y*z^3, y^3*z^2, y^2*z^3 ]
> MinimalPrimeComponents(BaseScheme(L4));
[
    Scheme over Rational Field defined by
    x
    y,
    Scheme over Rational Field defined by
    x
    z,
    Scheme over Rational Field defined by
    y
    z
]
> [ RationalPoints(Z) : Z in $1 ];
[
    {@ (0 : 0 : 1) @},
    {@ (0 : 1 : 0) @},
    {@ (1 : 0 : 0) @}
]
```

The linear system $L4$ has sections which are visibly those of $L3$ but with a single factor of z removed. It still has base locus but now that base locus comprises only points. Not surprisingly, it is exactly the three points which we imposed on curves in the first place.

`BasePoints(L)`

A sequence containing the basepoints of the linear system L if the base locus of L is finite dimensional.

`Multiplicity(L,p)`

The generic multiplicity of hypersurfaces of the linear system L at the point p.

112.16.2.3 Linear Algebra

`CoefficientSpace(L)`

The vector space corresponding to the linear system L whose vectors comprise the coefficients of the polynomial sections of L.

`CoefficientMap(L)`

The map from the polynomial ring that is the parent of the sections of the linear system L to the coefficient space of L. When evaluated at a polynomial f, this map will return vector of coefficients of f as a section of L.

`PolynomialMap(L)`

The map from the coefficient space of the linear system L to the polynomial ring that is the parent of the sections of L. When evaluated at a vector v, this map will return the polynomial section of L whose coefficients with respect to the basis of L are v.

`Complement(L,K)`

A maximal subsystem of the linear system L which does not contain any of the hypersurfaces of the linear system K.

`Complement(L,X)`

A maximal subsystem of the linear system L comprising hypersurfaces not containing X.

Example H112E56

In this example we show to define linear systems by referring to subspaces of a coefficient space. The explicit translation intrinsics between the linear algebra language and the linear system language let one 'see' what is happening in the background. We start by defining a linear system whose chosen sections are clearly not linearly independent.

```
> A<x,y> := AffineSpace(FiniteField(2),2);
> L := LinearSystem(A,[x^2-y^2,x^2,y^2]);
> VL := CoefficientSpace(L);
> VL;
KModule VL of dimension 2 over GF(2)
> W := sub< VL | VL.1 >;
> LinearSystem(L,W);
Linear system on Affine Space of dimension 2
Variables : x, y
with 1 section:
x^2
> phi := PolynomialMap(L);
> [ phi(v) : v in Basis(VL) ];
[
    x^2,
    y^2
]
```

Thus we see that Magma has chosen the obvious polynomial basis for the sections of L and disregarded the section $x^2 - y^2$.

`L meet K`

`Intersection(L,K)`

The linear system whose coefficient space is the intersection of the coefficient spaces of the linear systems L and K. An error is reported unless L and K lie in the same complete linear system.

`X in L`

Returns `true` if and only if the scheme X occurs among the hypersurfaces comprising the linear system L.

`f in L`

Returns `true` if and only if the polynomial f is a section of the linear system L. That is, `true` if and only if f is in the linear span of the basis of sections defining L.

`K subset L`

`IsSubsystem(L,K)`

Returns `true` if and only if the coefficient space of the linear system K is contained in that of the linear system L. An error is reported if L and K do not lie in a common linear system.

112.16.3 Linear Systems and Maps

The sequence of sections of a linear system may be used to construct a map from the projective space on which the sections are defined to another having the appropriate dimension. This is done directly using the `map` constructor as in Section 112.14. For example, if L is a linear system on some projective space P then the corresponding map can be created as follows.

```
> map< P -> Q | S >
>               where Q is ProjectiveSpace(BaseRing(P),#S-1)
>               where S is Sections(L);
```

There is not a proper inverse to this operation: there is no reason why a map should be determined by linearly independent polynomials. However, the system determined by the polynomials defining a map is still important. It is sometimes called the *homoloidal system* of the map.

```
Pullback(f,L)
```

The linear system f^*L on the domain of the map of schemes f where L is a linear system on the codomain of f. This requires care when f is not a regular map: it really produces the system of homaloids, that is, the substitution of the map equations into the linear system's sections.

112.17 Divisors

This section contains functionality for working with divisors on varieties (integral schemes defined over a field) of dimension greater than one. Currently, divisors can only be created on projective schemes X and there is also a restriction that X is ordinary projective for many of the less formal intrinsics that rely on the coherent sheaf code. In a number of places it is also required that a divisor D is Cartier (always true if X is non-singular), which we currently cannot check. There are also a number of intrinsics that are specific to surfaces.

Integral divisors are represented as differences of effective divisors, which are represented as subschemes of the scheme they live on. Factorisation into multiples of irreducibles can also be performed and the result is stored once calculated. It is also possible to work with **Q**-divisors. These are represented internally as factorisations with rational multiplicity of components.

The package of divisor functions is at and early stage and a number of the intrinsics are not as general as they could be and/or could be made more efficient. However, it is useful functionality that seems worth exporting now. There is much further work still to be done.

112.17.1 Divisor Groups

As for curves, there is a divisor group object associated to a variety X, which is the parent of all divisors on X. It is of type `DivSch`.

```
DivisorGroup(X)
```

The divisor group of variety X.

```
Variety(G)
```

The variety of the divisor group G.

```
G1 eq G2
```

True if and only if the divisor groups $G1$ and $G2$ are for the same variety.

112.17.2 Creation Of Divisors

Divisors are of type `DivSchElt`. Internally, an integral, effective divisor D on variety X is stored as an ideal which defines D as a subscheme of X. A general divisor is represented internally in partially factored form as a list of pairs of ideals and rational multiplicities $[\langle I_i, m_i\rangle]$ which represents the **Q**-rational (integral, if all m_i are integers) divisor $\sum_i m_i * D(I_i)$, where $D(I_i)$ is the effective divisor on X defined by the ideal I_i. An integral, effective divisor may also have a factorisation stored. The internal factorisation can change over time with the I_i being decomposed into products of larger ideals. When the I_i are all prime ideals, we say that the factorisation is a prime factorisation of D.

This section contains the basic creation functions for divisors.

```
Divisor(X,f)
Divisor(X,f)
Divisor(X,f)
```

These create the integral divisor on X defined by a single global element f. In the first two cases, f is an element of the function field (or the field of fractions of the coordinate ring of) the projective ambient of X. The divisor is non-effective (unless it is zero): the divisor of zeroes of f minus its divisor of poles. In the last case f should be a homogeneous polynomial in the coordinate ring of the ambient and the divisor is the effective divisor defined by the subscheme of X whose ideal is generated by the ideal of X and f.

```
Divisor(X,Q)
Divisor(X,Y)
Divisor(X,I)
```

`CheckSaturated`	BoolElt	*Default :* `false`
`CheckDimension`	BoolElt	*Default :* `false`
`UseCodimensionOnePart`	BoolElt	*Default :* `false`

These three intrinsics define an integral, effective divisor on projective variety X defined by an ideal or subscheme. For the first, Q is a sequence of elements in the coordinate ring of X and it is equivalent to passing the ideal generated by the ideal of X and Q. For the second, Y is a subscheme of X that should define an effective divisor on X. For the third, I is an ideal in the coordinate ring of the ambient of X, whose saturation J should contain the ideal of X. The effective divisor is given by the closed subscheme of X whose ideal is J.

`CheckSaturated` can be set to `true` in the third case if it is known that I is already saturated or in the second case if it is known that the ideal of Y is already saturated. `CheckDimension` can be set to `true` in any of the cases if it is known that the subscheme defining the divisor is of pure codimension 1 in X. Otherwise this condition is checked with the following exception. In the third case, if parameter `UseCodimensionOnePart` is set to `true` the non-codimension 1 part of the ideal is ignored in creating the divisor.

```
HyperplaneSectionDivisor(X)
```

Creates a divisor given by a hyperplane section of projective variety X.

```
ZeroDivisor(X)
```

The zero divisor on variety X.

```
CanonicalDivisor(X)
```

A canonical divisor on variety X. X must be ordinary projective and should be a Gorenstein scheme for this to give a correct result. It uses the canonical sheaf on X and the next intrinsic.

```
SheafToDivisor(S)
```

S is a coherent sheaf that should be invertible (locally free, rank 1) on variety X. S is then isomorphic to $L(D)$ for a (Cartier) divisor D (defined up to rational equivalence) on X. Returns such a divisor D which is effective if possible.

```
RoundDownDivisor(D)
```

For an integral divisor D just returns D. For a non-integral ($\mathbf{Q}$-rational) divisor with a factorisation into sum of rational multiples of prime components, returns the divisor of the integer multiple sum of primes given by rounding down all of the original rational coefficients.

```
RoundUpDivisor(D)
```

For an integral divisor D just returns D. For a non-integral ($\mathbf{Q}$-rational) divisor with a factorisation into sum of rational multiples of prime components, returns the divisor of the integer multiple sum of primes given by rounding up all of the original rational coefficients.

```
FractionalPart(D)
```

Returns $D-$ `RoundDownDivisor`(D).

```
IntegralMultiple(D)
```

Finds a positive integer N such that $E = N * D$ is an integral divisor. Returns E and N. Doesn't attempt to find the smallest possible N by analysing the full prime factorisation of D.

112.17.3 Ideals and Factorisations

Divisors are stored in ideal or factored form as described in the introduction to the last section. This section contains functions related to these representing structures.

```
Ideal(D)
```

Returns the defining ideal for an effective, integral divisor D.

```
Support(D)
```

The subscheme of the variety of effective $\mathbf{Q}$-divisor D that gives its support.

```
IdealOfSupport(D)
```

The ideal in the coordinate ring of the ambient of the variety of the effective $\mathbf{Q}$-divisor D that defines it's support.

```
SignDecomposition(D)
```

The decomposition of D into two effective divisors A and B such that $D = A - B$. A and B are returned. Note that they are not guaranteed to be relatively prime for this intrinsic.

```
IdealFactorisation(D)
```

Returns the current stored factorisation of D as a sequence of pairs of ideals and rational multiplicities.

```
CombineIdealFactorisation(~D)
```

Simplify the current ideal factorisation of D by combining terms with the same ideal.

```
ComputeReducedFactorisation(~D)
ReducedFactorisation(D)
```

Replace the ideal factorisation of D with an equivalent reduced factorisation where all ideals occurring are primary. The first intrinsic just does the replacement internally. The second intrinsic also returns the result.

```
ComputePrimeFactorisation(~D)
```

```
PrimeFactorisation(D)
```

Replace the ideal factorisation of D with an equivalent prime factorisation where all ideals occurring are prime. The first intrinsic just does the replacement internally. The second intrinsic also returns the result.

```
Multiplicity(D,E)
```

The multiplicity of prime divisor E in divisor D.

112.17.4 Basic Divisor Predicates

```
IsZeroDivisor(D)
```

Returns whether D is the zero divisor.

```
IsIntegral(D)
```

Returns whether D is an integral divisor. If this isn't immediately obvious from the current factorisation, will convert to a prime factorisation and try to combine terms.

```
IsEffective(D)
```

Returns whether D is an effective divisor. If this isn't immediately obvious from the current factorisation, will convert to a prime factorisation and try to combine terms.

```
IsPrime(D)
```

Returns whether D is a prime divisor.

```
IsFactorisationPrime(D)
```

Returns whether the current factorisation of D is a prime factorisation (i.e. all ideals occurring are prime).

```
IsDivisible(D)
```

Returns whether D is integral and divisible as an integral divisor by an integer $n > 1$. If so, also returns the maximum such n.

112.17.5 Arithmetic of Divisors

```
D1 + D2
D1 + D2
D1 + D2
D1 - D2
D1 - D2
D1 - D2
-D
```

Addition, subtraction and unitary minus on divisors. For addition and subtraction, one argument may be a toric divisor whose toric variety is the variety of the scheme divisor.

```
n * D
r * D
```

Multiplication of a divisor D by an integer n or rational number r. Note that the multiplication by r is the only current primitive method for constructing non-integral divisors.

```
D1 eq D2
```

Returns whether divisors $D1$ and $D2$ lie on the same variety and are equal.

112.17.6 Further Divisor Properties

More complicated predicates on divisors.

```
IsCanonical(D)
```

Returns whether D is a canonical divisor by testing whether its associated sheaf is isomorphic to the canonical sheaf. The variety of D must be ordinary projective here and should be Gorenstein.

```
IsAnticanonical(D)
```

Returns whether D is an anticanonical divisor by testing whether its associated sheaf is isomorphic to the dual of the canonical sheaf. The variety of D must be ordinary projective here and should be Gorenstein.

```
IsCanonicalWithTwist(D)
```

Returns whether D is the sum of a hypersurface divisor of degree d and a canonical divisor by testing whether its associated sheaf is isomorphic to a twist of the canonical sheaf. If so, also returns d. The variety of D must be ordinary projective here and should be Gorenstein.

`IsPrincipal(D)`

Returns whether D with variety X is a principal divisor and, if so, also returns an element f of the function field of the ambient of X such that $D = div(f)$. X should be ordinary projective and D a Cartier divisor here. Uses the Riemann-Roch space of D.

`IsLinearlyEquivalent(D,E)`

Returns whether two divisors D and E on variety X are linearly equivalent and, if so, also returns an element f of the function field of the ambient of X such that $D = E + div(f)$. Uses `IsPrincipal` for the difference between the two divisors, so X must be ordinary projective.

`BaseLocus(D)`

`IsBasePointFree(D)`

`IsMobile(D)`

The first intrinsic computes the base locus of the linear system $|[D]|$ (i.e. the reduced intersection of all effective divisors in the linear system) where $[D]$ is the round down of D . This uses the Riemann-Roch space of $[D]$, which means that this divisor has to be Cartier and the variety X of D has to be ordinary projective.

The second intrinsic returns whether this base locus is empty and the third whether it is of codimension at least two in X (i.e. there are no common divisor components to the full linear system). X and D obviously have to satisfy the same conditions.

`IntersectionNumber(D1,D2)`

$D1$ and $D2$ are divisors on a variety X which must be of dimension 2. One of the two divisors is assumed to be Cartier. Computes the intersection pairing number $D1.D2$.

`SelfIntersection(D)`

The intersection number of D with itself. D and X must satisfy the conditions for the last intrinsic.

`Degree(D)`

`Degree(D,H)`

D (resp. D and H) lies on a variety X of dimension 2. The first computes the intersection number of D with respect to a hyperplane divisor. The second computes the intersection number of D with H (so is just equivalent to `IntersectionNumber(D,H)`).

`IsNef(D)`

D should be a $\mathbf{Q}$-Cartier divisor on a projective surface X. Currently, it also has to be effective. Returns whether D is a nef divisor: i.e. whether it has non-negative intersection with all effective divisors on X.

```
IsNefAndBig(D)
```

D and X are as above. D must be effective. Returns whether D is a nef divisor AND has positive self-intersection.

```
NegativePrimeDivisors(D)
```

D and X again should satisfy the same conditions as in `IsNef`. Returns a sequence of prime divisor components of D which have negative intersection with D.

```
ZariskiDecomposition(D)
```

D and X again should satisfy the same conditions as in `IsNef`. Returns a pair of $\mathbf{Q}$-divisors P and N such that $D = P + N$, P is nef and N has negative-definite support (i.e. the intersection pairing on its prime components is negative definite).

112.17.7 Riemann-Roch Spaces

This section contains functions to compute and work with Riemann-Roch spaces for a divisor D. It is always assumed that X, the variety of D, is an ordinary projective space here. The Riemann-Roch space is the finite-dimensional subspace of the vector space (over the basefield) of rational functions on X consisting of zero and all $f \neq 0$ such that $D + div(f)$ is an effective divisor. Thus, the Riemann-Roch space of D is the same as the Riemann-Roch space of $[D]$, `RoundDownDivisor(D)`. Because the sheaf code is used (or a slight variant for non-effective divisors), it is also required that $[D]$ is Cartier for all relevant intrinsics.

```
Sheaf(D)
```

The invertible sheaf corresponding to the divisor class of divisor D. If D is not integral, its round down $[D]$ is used. This divisor must be Cartier and its variety X must be ordinary projective. Is effectively the same function as in the Sheaf package when D is effective and uses a slight variant otherwise.

```
RiemannRochBasis(D)
RiemannRochSpace(D)
```

The first returns a basis of the Riemann-Roch space of the round down $[D]$ of divisor D (as a sequence of elements in the function field F of the ambient of D's variety, X). The second returns the Riemann-Roch space as an abstract vector space V over the base field along with a map from V to F. X must be ordinary projective and $[D]$ Cartier. If D is effective, this is the same as using the coherent sheaf function that computes the associated invertible sheaf and Riemann-Roch basis. If D is non-effective a slight variant is used.

```
RiemannRochCoordinates(f,D)
```

Returns whether f can be coerced into the function field of the ambient of D's variety X and whether f then lies in the Riemann-Roch space of D. If so also returns the coordinates of f with respect to the basis of the Riemann-Roch space returned by `RiemannRochBasis(D)`. As usual, X must be ordinary projective and $[D]$, the round down of D, must be Cartier.

```
IsLinearSystemNonEmpty(D)
```

Returns whether there is an effective divisor linearly equivalent to D and, if so, returns such a divisor. Uses the Riemann-Roch space of D. The conditions on D and X, its variety, are as for the preceding intrinsics.

112.18 Isolated Points on Schemes

There is now experimental code to try to find isolated points of schemes in $\mathbf{A}(Q)^n$ that are defined by n or more equations. There is no restriction that the scheme itself must be of dimension 0, and in many applications, there is an uninteresting (or degenerate) variety of positive dimension, with the isolated points being the ones of interest.

The first technique to do this is to find points locally modulo some prime, at which the Jacobian matrix is of maximal rank. A separate procedure then lifts such points via a Newton method, and tries to identify them (via LLL) in a number field. If the degree is known (or suspected), this information can be passed to the lifting function, and otherwise it will try a generic method that applies LLL for degrees $3 \cdot 2^n$ and $4 \cdot 2^n$ as n increases, expecting to catch other solutions via a factorization into a smaller field (for instance, a degree 6 solution will show up in degree 8 as the expected minimal polynomial multiplied by a more-or-less random quadratic one).

The running time of such a process is often dominating by the searching for local points, and so various methods can be used to pre-condition the system. Firstly, variables which appear as a linear monomial in one of the equations can be (iteratively) eliminated. Already it is not clear what the best order is, so the user can specify it. In given examples, it is not untypical for later steps to take 10 times as long due to a different choice of eliminations, so some experimenting with this parameter can be useful. Secondly, resultants can be used to try to eliminate more variables, perhaps concentrating on those which appear to a small degree in the equations. By default, if a variable appears to degree 2 or less in all the equations, a resultant step will be applied. The resulting equations can be quite complicated (taking many megabytes to represent them), but it is still often faster to loop over p^{V-1} possible local points compared to p^V. The resultants can introduce extraneous solutions, which are checked when undoing the modifications to the system of equations. An error can occur if the variable elimination reduces the number of equations below the dimension.

Another concern for running time is in the order of the above operations. For instance, computing the resultants can be time-consuming in some examples, and perhaps doing so modulo p for the primes of interest might be a superior method in some cases. A second example is that recognising points over the number fields in question might be faster with a reduced system and then undoing the resultants and linear eliminations in the number field, though the lifting might be slower in that case. The algorithm as implemented tries to handle the general case well.

Note that the scheme given to the IsolatedPoints routines might also have components of positive dimension, but the techniques here will only work to find the points that do not lie on them.

```
LinearElimination(S)
```

`EliminationOrder`	SeqEnum	*Default :* [ ]

Given a scheme, iteratively eliminate variables that appear strictly linearly (that is, as a monomial times a constant) in some equation. The `EliminationOrder` vararg allows an ordering to be specified. The results can vary drastically when the order is changed. The returned value is a map from the resulting scheme to the input scheme, with an inverse.

```
IsolatedPointsFinder(S,P)
```

`LinearElimination`	SeqEnum	*Default :* [ ]
`ResultantElimination`	SeqEnum	*Default :* [ ]
`FactorizationInResultant`	BoolElt	*Default :* `true`

Given a affine n-dimensional scheme defined over the rationals by at least n equations, and a sequence of primes, try to find liftable points of the scheme modulo the primes. The variables given in the `LinearElimination` vararg will be eliminated in that order. If this is non-empty, an automatic procedure will be applied. Similarly with `ResultantElimination`. Finally, by default, computed resultants have `SquarefreeFactorization` applied to them (and repeated factors removed), and this can be turned off.

```
IsolatedPointsLifter(S,P)
```

`LiftingBound`	RngIntElt	*Default :* 10
`DegreeBound`	RngIntElt	*Default :* 32
`OptimizeFieldRep`	BoolElt	*Default :* `false`
`DegreeList`	SeqEnum	*Default :* [ ]

Given a affine n-dimensional scheme defined over the rationals by at least n equations, and a sequence of finite field elements giving a point (on the scheme) whose Jacobian matrix is of maximal rank, attempt to lift the point via a Newton method and recognise it over a number field. The function returns `false`if a point was not found, and else returns both `true`and the point that was found.

The `LiftingBound` vararg determines how many lifting steps to use, with the default being 10, so that approximately precision $p^{2^{10}}$ is obtained. The `DegreeBound` is a limit on how high of a degree of field extension to check for solutions. The method used only checks for some of the degrees via LLL, as usually smaller degrees will show up in factors. The `DegreeList` vararg can also be used for this, perhaps when the degree of the solution in question is known. The practical limit is likely around 50 in the degree. Finally, there is the `OptimizeFieldRep` boolean vararg, which determines whether the number field obtained will have its optimised representation computed.

```
IsolatedPointsLiftToMinimalPolynomials(S,P)
```

`LiftingBound`	RngIntElt	*Default :* 10
`DegreeBound`	RngIntElt	*Default :* 32
`DegreeList`	SeqEnum	*Default :* [ ]

This works as with `IsolatedPointsLifter`, but instead of trying to find a common field containing all the coordinates, simply returns a minimal polynomial for each one.

Example H112E57

This example follows an idea of Elkies to construct "large" integral points on elliptic curves. Let X, Y, A, B be polynomials in t, and let $Q(t)$ be quadratic. The idea is that

$$Q(t)Y(t)^2 = X(t)^3 + A(t)X(t) + B(t)$$

will have infinitely many specialisations with $Q(t)$ square (via solving a Pell equation), and thus yield, perhaps after scaling to clear denominators, integral points on the resulting curves. If the degree of A, B is small enough compared to that of X, the resulting specialisation will be quite notable. However, one must avoid the cases where the resulting discriminant $4A(t)^3 - 27B(t)^2$ is zero, as these points do not yield an elliptic curve. It turns out that such points contribute a positive-dimensional component to the solution space, and we simply want to ignore such solutions in general.

The first case of interest is when the degrees of (X, Y, A, B) are $(4, 5, 0, 1)$, solved by Elkies in 1988. One way of parametrising this, taking into account possible rational changes of the t-variable to simplify the system, is:

$$X(t) = t^4 + t^3 + x_2 t^2 + x_1 t + x_0, Q(t) = t^2 + q_1 t + q_{,}0$$

$$Y(t) = t^5 + y_3 t^3 + y_2 t^2 + y_1 t + y_0, A(t) = a_0, B(t) = b_1 t + b_0.$$

Then we get 12 equations from requiring that the 0th to 11th degree coefficients of $X^3 + AX + B - QY^2$ all vanish. It turns out that the desired solution is rational in this case, so that (almost) any prime will suffice. The solution can then be lifted – as a final step (particular to this problem), one would have to scale the resulting point so as to ensure that $Q(t)$ represents squares.

```
> K := Rationals();
> R<a0,b0,b1,q0,q1,x0,x1,x2,y0,y1,y2,y3> := PolynomialRing(K,12);
> _<t> := PolynomialRing(R);
> X := t^4+t^3+x2*t^2+x1*t+x0; Y := t^5+y3*t^3+y2*t^2+y1*t+y0;
> Q := t^2+q1*t+q0; A := a0; B := b1*t+b0;
> L := X^3+A*X+B-Q*Y^2;
> COEFF:=[Coefficient(L,i) : i in [0..11]];
> S := Scheme(AffineSpace(R),COEFF);
```

For this example, we simply call `IsolatedPointsFinder` directly. Alternatively, we could first use `LinearElimination` if desired.

```
> PTS:=IsolatedPointsFinder(S,[13]); PTS; // 13 is a random choice
[* [ 11, 1, 7, 6, 3, 1, 6, 11, 0, 3, 4, 2 ] *]
> b, sol := IsolatedPointsLifter(S,PTS[1]); sol;
(216513/4096, -3720087/131072, 531441/8192,
 11/4, 3, 311/64, 61/8, 9/2, 715/64, 165/16, 77/16, 55/8)
> _<u>:=PolynomialRing(Rationals());
> X := Polynomial([Evaluate(c,Eltseq(sol)) : c in Coefficients(X)]);
> Y := Polynomial([Evaluate(c,Eltseq(sol)) : c in Coefficients(Y)]);
> Q := Polynomial([Evaluate(c,Eltseq(sol)) : c in Coefficients(Q)]);
> A := Evaluate(A,Eltseq(sol));
> B := Polynomial([Evaluate(c,Eltseq(sol)) : c in Coefficients(B)]);
> assert X^3+A*X+B-Q*Y^2 eq 0;
> Q; // note that Q does not represent any squares, but 2*Q(1/2)=9
u^2 + 3*u + 11/4
> B; // also need to clear 2^17 from denominators
531441/8192*u - 3720087/131072
> POLYS := [2^7*X, 2^9*Y, 2^3*Q, 2^14*A, 2^21*B]; // 2^21 in each term
> [Evaluate(f,u/2) : f in POLYS];
[
    8*u^4 + 16*u^3 + 144*u^2 + 488*u + 622,
    16*u^5 + 440*u^3 + 616*u^2 + 2640*u + 5720,
    2*u^2 + 12*u + 22,
    866052,
    68024448*u - 59521392
]
```

As noted by Elkies, one can clean up the final form of the solution if desired, via rational transformations of the u-variable. Since $Q(1) = 2 + 12 + 22 = 36$, the theory of the Pell equation tells us that there are infinitely many integers u such that $Q(u)$ is integral, and these all give integral points on a suitable elliptic curve.

There are only four choices of (X, Y, A, B) degrees that give the "largest" possible integral points via this method. The second case, of degrees $(6, 8, 1, 1)$ has a solution over a quartic number field, and the third case, of degrees $(8, 11, 1, 2)$ has a nonic solution. Both of these were found by the methods of this section, the former taking only a couple of minutes.

Example H112E58

Another application of the isolated points rotine is to compute Belyi maps, one instance of which is in finding solutions to a polynomial version of Hall's conjecture, concerning how small the degree of $X(t)^3 - Y(t)^2$ can be (if the difference is nonzero). The result is that the degree must be at least $1 + \deg(X)/2$, and there are (up to equivalence) finitely many polynomials of that degree, the count of which can be described in terms of some combinatorics, or in terms of simultaneously conjugacy classes of cycle products of given types in a symmetric group.

In this example, we compute the solution for the case where X has degree 12. It turns out that there are 6 solutions in this case, lying over a sextic number field (we of course ignore "solutions" with $X(t)^3 = Y(t)^2$, though similar to the previous example, they contribute a positive-dimensional component of the solution set). The finding of a suitable local point is not particularly easy, so we just note that

$$X(t) \equiv t^{12} + 14t^{10} + 14t^9 + 9t^8 + 6t^7 + 4t^6 + 7t^5 + 6t^4 + 15t^3 + 7t^2 + 3t + 10$$

gives a solution modulo 17, with $Y(t)$ being (of course) the approximate square root of $X(t)^3$. We shall lift this in a way that keeps the t^{11} as zero and the t^9 and t^{10} coefficients as equal (these are from preliminary transformations of the t-parameter that can be applied to the system).
As noted above, it might be easier to remove the y-variables "by hand", and then undo the linear eliminations in the resulting sextic number field. We chose here to work directly. A theorem of Beckmann says that the number field we obtain can only be ramified at primes less than 36.

```
> SetVerbose("IsolatedPoints",1);
> XVARS := ["x"*IntegerToString(n) : n in [0..9]];
> YVARS := ["y"*IntegerToString(n) : n in [0..17]];
> P := PolynomialRing(Rationals(),28);
> AssignNames(~P,XVARS cat YVARS);
> _<t> := PolynomialRing(P);
> Y := &+[P.(i+11)*t^i : i in [0..17]]+t^18;
> X := &+[P.(i+1)*t^i : i in [0..9]]+(P.10)*t^10+t^12;
> Xpt := [GF(17)|10,3,7,15,6,7,4,6,9,14];
> pt := Xpt cat [0 : i in [11..28]];
> FF := GF(17); _<u> := PolynomialRing(FF);
> Xv := Polynomial([FF!Evaluate(c,pt) : c in Coefficients(X)]);
> Xv3 := Xv^3; Yv := u^18;
> for d:=17 to 0 by -1 do // ApproximateSquareRoot
>     Yv:=Yv+Coefficient(Xv3,d+18)/2*u^d; Xv3:=Xv^3-Yv^2; end for;
> Yv^2-Xv^3; // must be degree 7 or less
8*u^7 + 11*u^5 + 10*u^4 + 3*u^3 + 3*u^2 + 11*u + 4
> pt := Xpt cat [Coefficient(Yv,d) : d in [0..17]];
> SYS := [Coefficient(X^3-Y^2,d) : d in [8..35]]; // 28 vars
> S := Scheme(AffineSpace(P),SYS);
> b, sol := IsolatedPointsLifter(S,pt : LiftingBound:=12, DegreeBound:=10);
> K := OptimisedRepresentation(Parent(sol[1]) : PartialFactorisation); K;
Number Field with defining polynomial
  y^6 - y^5 - 60*y^4 - 267*y^3 - 514*y^2 -  480*y - 180 over the Rationals
> Factorization(Discriminant(Integers(K)));
[ <2, 2>, <5, 1>, <13, 1>, <29, 5> ]
```

An alternative of completing the computation is to first use `LinearElimination` before applying `IsolatedPointsLifter`. In any event, the computation of the local point (which we simply assumed to be given) would be the dominant part of the running time.

```
> mp := LinearElimination(S); // a few seconds to evaluate scheme maps
> rmp := // reduced map
>  map<ChangeRing(Domain(mp),GF(17))->ChangeRing(Codomain(mp),GF(17))
>     | DefiningEquations(mp),DefiningEquations(Inverse(mp)) : Check:=false>;
> PT := Inverse(rmp)(Codomain(rmp)!(pt));
```

The `IsolatedPointLifter` can now be called on `Domain(mp)` and `Eltseq(PT)` (with varargs if desired), and then the result can be mapped back to the original scheme `S` via `mp`. It is a bit hairy to do this directly, as scheme maps do not naturally deal with finite field inputs in all cases. Due to the way that `IsolatedPointsLifter` uses to choose which coordinate to try to recognise first, it could also be slower in the end.

Example H112E59

Here is an example where finding the common field is quite difficult, but finding minimal polynomials for all the coordinates is rather easy. First, a somewhat generic random scheme in $\mathbf{P}^3$ is chosen, such that each variable appears no more than linearly. This has degree less than 4!, and in the example chosen, it has degree 22. Then two local points are found modulo 5. These are then passed to the lifting function, which returns the desired solution.

```
> P<w,x,y,z> := AffineSpace(Rationals(),4);
> f1 := w*x*y - 6*w*x - 7*w*y*z + w*y - 6*w*z - 3*x*y + y + 6*z;
> f2 := 10*w*x*y*z - 4*w*y*z + 2*w*y - 9*w - x*y*z - 10*x*z + y*z - 7*y;
> f3 := 10*w*x*y*z - 6*w*x*y + 8*w*x*z - 4*w*y*z - 6*w*z - x*z + 9*x + 8*y;
> f4 := 6*w*x*y*z + 3*w*x*z + 19*w*y*z - 7*w*z + 8*x*y*z - 2*x*z + 6;
> S := Scheme(P,[f1,f2,f3,f4]);
> SetVerbose("IsolatedPoints",1);
> PTS := IsolatedPointsFinder(S,[5]);
> Degree(S);
22
> b,POLYS := IsolatedPointsLiftToMinimalPolynomials
>                      (S,PTS[1] : DegreeBound:=22,LiftingBound:=10);
> POLYS[1];
18124035687220989600*x^22 + 62977055844929678832*x^21 +
 65273363651442356128*x^20 + 81271204075826455992*x^19 +
 130701369600138969680*x^18 - 285376384061267841622*x^17 -
 802166956118815471654*x^16 + 253325444790327996845*x^15 -
 1266591733002155213172*x^14 + 25113861844403230090*x^13 +
 506530967406804631482*x^12 - 1323179973699695447463*x^11 +
 1605685921502538803112*x^10 - 1318315736155520576802*x^9 +
 949649129582958459958*x^8 - 527441332544171338490*x^7 +
 254463684049866607512*x^6 - 100039189198577581440*x^5 +
 26014411295686475856*x^4 - 3177984195514332576*x^3 -
 1852946687180290752*x^2 + 971825485320437760*x - 88506566917263360
```

All of the four polynomials in `POLYS` look approximately like this, and all should determine the same field, but it is difficult to find suitable isomorphisms between them, let alone find an `OptimisedRepresentation`.

In fact, the Gröbner basis machinery is superior for this purpose. Writing one of the coordinates in terms of the other (so as to get the field generators in terms of each other) would necessitate quite high precision in the p-adic lifting to recognise the coefficients, likely $5^{2^{14}}$ or more (this takes about 5 minutes). The Gröbner basis method, which recognises one coordinate and then back-substitutes into the resulting equations, solving them algebraically, takes only about 15 seconds.

```
> time V := Variety(Ideal(S),AlgebraicClosure()); // about 15s
> MinimalPolynomial(V[22][4]); // deg 22, all coeffs about 25 digits
y^22 - 70869414518205839537/14232439756116709952*y^21 -
    606754258610022348837З/56929759024466839808*y^20 + [...]
    [...] + 166661449939161/177905496951458874 4
> MinimalPolynomial(V[22][3]); // deg 22, all coeffs about 25 digits
y^22 - 428567519465749893/68067993818308256*y^21 -
    2295929539688005961 5/1089087901092932096*y^20 + [...]
    [...] + 2469165405490441431/68067993818308256
> V[22][4]; // given simply as r22, all 22 conjugates are found
r22
> V[22][3]; // third coordinate in terms of the fourth
[output takes about 200 lines, involving 750-digit coordinates]
```

Another way to achieve the result is to plug the known coordinate into the system, and use Gröbner bases (or resultants, if possible) to solve it.

```
> K := NumberField(POLYS[1]); // first coordinate
> _<xx,yy,zz> := PolynomialRing(K,3);
> E := [Evaluate(e,[K.1,xx,yy,zz]) : e in DefiningEquations(S)];
> Variety(Ideal(E)); // about 2 seconds
[again a rather bulky output]
```

Both of these uses of Gröbner bases are somewhat specific to the simplicity of the case here, and in more difficult cases would likely be rather onerous. This example does exemplify that for a generic variety the Gröbner basis methods should be superior. The lifting methods are largely for cases where the problem has special structure.

Example H112E60

Here is an example from N. D. Elkies of a polynomial $f(x) \in K(x)$ for which $f(x) - t$ appears to have Galois group M_{23} over $K(t)$ (this is sometimes called the monodromy group of f). This involves trying to find $f = P_3 P_2^2 P_4^4 = P_7 P_8^2 - c$ for some polynomials P_d of degree d, for some constant c. Upon suitable reductions, one gets a system of 8 variables and equations. With a few additional considerations, the search space can be reduced a bit further. As noted by M. Zieve, the equation should have 4 solutions, and thus likely be in a quartic field K that is an extension of $\mathbf{Q}(\sqrt{-23})$. This is indeed the case.

```
> Q := Rationals();
> R<a2,c,b1,b2,c1,c2,c3,c4,d1,d2,d3,d4,d5,d6,d7,
>   e1,e2,e3,e4,e5,e6,e7,e8> := PolynomialRing(Q,(3-1)+2+4+7+8);
> _<t> := PolynomialRing(R);
> P3 := t^3 + a2*t + a2/(26/-27); // normalisation
> P2 := t^2 + b1*t + b2;
> P4 := t^4 + c1*t^3 + c2*t^2 + c3*t + c4;
> P7 := t^7 + d1*t^6 + d2*t^5 + d3*t^4 + d4*t^3 + d5*t^2 + d6*t + d7;
> P8 := t^8 + e1*t^7 + e2*t^6 + e3*t^5 + e4*t^4 +
>            e5*t^3 + e6*t^2 + e7*t + e8;
> Q := P3 * P2^2 * P4^4 - P7 * P8^2 - c;
> S := Scheme(AffineSpace(R),Coefficients(Q));
> SetVerbose("IsolatedPoints",1);
> v:=[GF(101) | 26, 1,  -26,21,  -19,-27,-22,8, // known point
>              -14,12,26,-3,-37,-43,-22,  44,-11,-13,-21,45,-45,32,46];
> b, pt := IsolatedPointsLifter
>          (S,v : DegreeList:=[4], LiftingBound:=15, OptimizeFieldRep);
> K := Parent(pt[1]);
> DefiningPolynomial(K);
y^4 - 2*y^3 - 10*y^2 + 11*y + 36
> Factorization(Discriminant(Integers(K)));
[ <3, 1>, <23, 3> ]
```

The above code lifts the given point to precision $101^{2^{11}}$, and recognises it in the field K. Next we can compute the polynomial $f(x)$ in question, and see that reductions (modulo 269, say) of $f(x)-i$ do indeed correspond to cycle structures of M_{23}. However, to prove that this really is the Galois group (over the function field) seems to require a more difficult monodromy calculation. A more theoretical (topological) construction was given by P. Müller in 1995, but did not explicitly produce f. It is also still an open question whether there is a polynomial over the *rationals* with Galois group M_{23}.

```
> X := P3 * P2^2 * P4^4;
> f := Polynomial([Evaluate(e,Eltseq(pt)) : e in Coefficients(X)]);
> p := 269;
> P := Factorization(p * Integers(K))[1][1]; assert Norm(P) eq p;
> _, mp := ResidueClassField(P);
> fp := Polynomial([mp(c) : c in Coefficients(f)]);
> D := [[Degree(u[1]) : u in Factorization(fp-i)] : i in [1..p]];
> Sort(SetToSequence(Set([Sort(d) : d in D | &+d eq 23])));
[
    [ 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2 ],
    [ 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3 ],
    [ 1, 1, 1, 2, 2, 4, 4, 4, 4 ],
    [ 1, 1, 1, 5, 5, 5, 5 ],
    [ 1, 1, 7, 7, 7 ],
    [ 1, 2, 2, 3, 3, 6, 6 ],
    [ 1, 2, 4, 8, 8 ],
    [ 1, 11, 11 ],
    [ 2, 7, 14 ],
    [ 3, 5, 15 ],
    [ 23 ]
]
> load m23; // G is M23
> C := [g : g in ConjugacyClasses(G) | Order(g[3]) ne 1];
> S := Set([CycleStructure(c[3]) : c in C]);
> Sort([Sort(&cat[[s[1] : i in [1..s[2]]] : s in T]) : T in S]);
[
    [ 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2 ],
    [ 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3 ],
    [ 1, 1, 1, 2, 2, 4, 4, 4, 4 ],
    [ 1, 1, 1, 5, 5, 5, 5 ],
    [ 1, 1, 7, 7, 7 ],
    [ 1, 2, 2, 3, 3, 6, 6 ],
    [ 1, 2, 4, 8, 8 ],
    [ 1, 11, 11 ],
    [ 2, 7, 14 ],
    [ 3, 5, 15 ],
    [ 23 ]
]
```

112.19 Advanced Examples

This section contains examples of the use of the scheme machinery that are broader than those brief illustrations of intrinsics in the main text. They show how these functions can be used in collaboration with one another to build computer experiments which back up mathematical intuition.

112.19.1 A Pair of Twisted Cubics

This example constructs a cluster as the intersection of two twisted cubics in space. It uses a pair of curves whose equations are very closely related. Their union admits an automorphism which interchanges the two curves, fixing the cluster.

Example H112E61

We start by making the two twisted cubics, $C1$ and $C2$, as the minors of a pair of 2×3 matrices. It is clear straight away that these curves are closely related; it is a shame that we lose the "format" of the equations, in fact. On the other hand, if one tries to make other interesting examples by such tricks, one does not automatically come up with something so slick (and Gorenstein).

```
> k := Rationals();
> P<x,y,z,t> := ProjectiveSpace(k,3);
> M1 := Matrix(CoordinateRing(P),2,3,[y,t,x,t,x,z]);
> M2 := Matrix(CoordinateRing(P),2,3,[y,x,t,x,t,z]);
> C1 := Scheme(P,Minors(M1,2));
> C2 := Scheme(P,Minors(M2,2));
> Z := Intersection(C1,C2);
> MinimalBasis(Z);
[
    x*t - y*z,
    x*z - t^2,
    x*y - t^2,
    -x^2 + z*t,
    -x^2 + y*t
]
```

Anyone knowing about Pfaffians can have fun trying to realise these equations as the five maximal Pfaffians of a skew-symmetric 5×5 matrix. Although this example is a bit degenerate, it is reasonable to think of it as a hyperplane section of an elliptic curve of degree 5 (living in $\mathbf{P}^4$) so the ideal of equations will be Gorenstein. Given the Buchsbaum–Eisenbud structure theorem for Gorenstein codimension 3 rings, we are not surprised to see this Pfaffian format. In this example we will settle for confirming that this scheme Z is a cluster of degree 5 and finding its support.

```
> IsCluster(Z);
true
> Degree(Z);
5
> IsReduced(Z);
true
> RationalPoints(Z);
{@ (1 : 1 : 1 : 1), (0 : 0 : 1 : 0), (0 : 1 : 0 : 0) @}
> HasPointsOverExtension(Z);
true
```

As expected, the scheme Z is zero-dimensional and has degree 5. Since it is reduced, its support will comprise five separate points over some extension of the base field. We locate these points by hand by considering the Gröbner basis of the ideal of Z. The last element of a lexicographical Gröbner basis usually suggests a field extension that is relevant to the scheme. So we extend the base field by roots of this polynomial and look for the support over that field.

```
> GB := GroebnerBasis(ChangeOrder(Ideal(Z),"lex"));
> GB[#GB];
z^3*t - t^4
> L<w> := ext< k | U.1^2 + U.1 + 1 > where U is PolynomialRing(k);
> RationalPoints(Z,L);
{@ (w : -w - 1 : -w - 1 : 1), (-w - 1 : w : w : 1),
(1 : 1 : 1 : 1), (0 : 1 : 0 : 0), (0 : 0 : 1 : 0) @}
> HasPointsOverExtension(Z,L);
false
```

The final line confirms that we have found all the points of Z. That was already clear since Z has degree 5 and we see five points, but in other cases, especially when the cluster is not reduced, it might not be so obvious.
Now we look at the union of the two twisted cubics.

```
> C := Union(C1,C2);
> C;
Scheme over Rational Field defined by
x^3 - x*y*t - x*z*t + t^3
-x*t + y*z
```

This curve C is a $2, 3$ complete intersection, numerology that is familiar from canonical curves of genus 4. We already know that C is not such a curve since it has two components. Indeed, we already know that these components are nonsingular and meet in five points. Clearly these points must be singular points of C.

```
> SC := SingularPointsOverSplittingField(C);
> SC;
{ (1 : 1 : 1 : 1), (-r2 - 1 : r2 : r2 : 1), (0 : 0 : 1 : 0),
(-r1 - 1 : r1 : r1 : 1), (0 : 1 : 0 : 0) }
> Ring(Universe(SC));
Algebraically closed field with 2 variables
Defining relations:
[
    r2^2 + r2 + 1,
    r1^2 + r1 + 1
]
```

Magma has automatic Gröbner basis based machinery for working in the algebraic closure of the rationals (the so-called *D5 method*). Here we see it in action. The roots that we made explicitly when computing with Z are the new symbols $r1$ and $r2$ — they are the two conjugate roots of the quadratic equation list as the 'Defining relations'. Since $r1 \neq r2$, we see that the singular points really are the points of Z as expected.

From the definition of the matrices $M1$ and $M2$ we can see that the union and intersection of $C1$ and $C2$ should be invariant under the automorphism of P which exchanges x and t. We realise that automorphism here and confirm what we expect by comparing various ideals.

```
> phi := iso< P -> P | [t,y,z,x],[t,y,z,x] >;
> IsAutomorphism(phi);
true
> Ideal(C2) eq Ideal(phi(C1));
true
> Z eq phi(Z);
true
> Ideal(Z) eq Ideal(phi(Z));
true
```

Note that the basic equality test 'eq' for schemes returns `true` in the penultimate line, even though the two arguments were created independently.

The five points of Z obviously have Sym_5 as their permutation group (or $\mathrm{Sym}_2 \times \mathrm{Sym}_3$ over the rationals). How much of that is realised by automorphisms of the union C? We try to realise some elements of this symmetric group.

```
> S5 := SymmetricGroup(5);
> QL := RationalPoints(Z,L);
> rho := S5 ! [ Index(QL,phi(p)) : p in QL ];
> rho;
(1, 2)
```

Of course, this permutation is simply the action of the Galois group of L.

```
> GaloisGroup(L);
Permutation group acting on a set of cardinality 2
```

```
    (1, 2)
```

We make another automorphism: using C explicitly in the constructor ensures that the image of the map is contained in C.

```
> psi := iso< C -> C | [x,z,y,t],[x,z,y,t] >;
> eta := S5 ! [ Index(QL,psi(p)) : p in QL ];
> eta;
(4, 5)
> G := sub< S5 | rho,eta >;
> #G;
4
```

Since these two permutations commute and the small collection of five points is already partitioned by a Galois group action, this example is too simple to use Magma's substantial group theory machinery. But one can imagine at this stage finding complicated elements of G and realising them by compositions of the easily recognised automorphisms ρ and η.

112.19.2 Curves in Space

In this example, we construct something that we know is an elliptic curve in space. The point is to realise that within Magma by making a new curve of the right type and understanding the translation between the two types, at least to some degree. Something very similar would also works for the canonical models of curves of genus 4, although one has to take care handling the image of the natural projection.

Example H112E62

The first thing to do is to make a curve in space and to choose a nonsingular rational point on that curve. The question of whether or not a rational point is part of the input or part of the algorithm is always tricky since finding good points is often the heart of a problem. That is certainly the case here, so we do not pretend that this is a particularly powerful example.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Scheme(P,[x*y-z*t,x^2 + 2*z^2 - y*t]);
> Dimension(X);
1
> IsNonsingular(X);
true
> p := X ! [0,1,0,0];
```

Next we simply project from this given point p.

```
> Y,pr,q := ProjectionFromNonsingularPoint(X,p);
> bool,C := IsCurve(Y);
> bool;
true
> q := C ! q;
> q;
(0 : 1 : 0)
```

```
> Degree(C);
3
> IsNonsingular(C);
true
> P2<a,b,c> := Ambient(C);
> C;
Curve over Rational Field defined by
a^3 + 2*a*b^2 - b*c^2
```

Since there was a conic (two, in fact) among the equations of the scheme X, the projection from p is necessarily birational to a plane curve. And since p is a nonsingular point, it has a definite rational image point on the projection which is called q above. Since we know that X has genus 1 (as an external fact) and that the projection is birational we already know that the image curve C is the plane elliptic curve we desire. (It is interesting to try this calculation with a curve of higher genus, like the canonical model of a curve of genus 4.)
But we have made no effort to find a good model for C. At this point we can use Magma intrinsics to find a better model for C since we have the rational point q lying on C.

```
> EllipticCurve(C,q);
Elliptic Curve defined by y^2 = x^3 + 32*x over Rational Field
Mapping from: Crv: C to Elliptic Curve defined by y^2 = x^3 + 32*x over
Rational Field given by a rule
Mapping from: Elliptic Curve defined by y^2 = x^3 + 32*x over Rational Field to
Crv: C given by a rule
```

The result is a very nice model of an elliptic curve in Weierstraß form. The mapping types returned by this function are not yet fully integrated Scheme maps. But this will be added to Magma in due course, after which computations can be done on the good elliptic model and related to the original scheme X.

112.20 Bibliography

[**BC04**] W. Bosma and J. Cannon, editors. *Discovering Mathematics with Magma.* Springer-Verlag, Heidelberg, 2004.

[**Bru04**] Nils Bruin. Some ternary Diophantine equations of signature $(n, n, 2)$. In Bosma and Cannon [BC04].

[**Har77**] Robin Hartshorne. *Algebraic Geometry, GTM 52.* Springer, ASpringer, 1977.

[**MSS96**] J. R. Merriman, S. Siksek, and N. P. Smart. Explicit 4-descents on an elliptic curve. *Acta Arith.*, 77(4):385–404, 1996.

[**Rei88**] Miles Reid. *Undergraduate Algebraic Geometry.* CUP, Cambridge, 1988.

[**Rei97**] Miles Reid. Chapters on Algebraic Surfaces. In J. Kollár, editor, *Complex algebraic varieties, IAS/Park City Mathematics Series 3*, pages 1 – 154. AMS, Providence R.I., 1997.

[**vdE00**] Arno van den Essen. *Polynomial automorphisms and the Jacobian conjecture*, volume 190 of *Progress in Mathematics.* Birkhaeuser, ABirkhaeuser, 2000.

113 COHERENT SHEAVES

Chapter 113
COHERENT SHEAVES

113.1 Introduction

This chapter describes the Magma functionality for working with coherent sheaves on ordinary projective schemes. The emphasis in this initial version is on invertible sheaves and on computing associated cohomological invariants and explicit divisor maps. Important examples include canonical and anticanonical maps and adjunction maps on varieties of arbitrary dimension. The tools provided in Magma enable the user to compute these in a general and reasonably efficient way. There is also functionality for computing an invertible sheaf corresponding to the class of an effective Cartier divisor given as a closed subscheme as well as a basis for the Riemann-Roch space of that divisor as ambient rational functions. The correspondence between divisors (or their classes) and invertible sheaves will be expanded in later releases. A standard reference for the definition and basic properties of coherent sheaves on Noetherian schemes is Section 5, Chapter II of [Har77].

The package is based on Magma's functionality for graded modules over polynomial rings and relies heavily on Gröbner basis computations. A coherent sheaf is represented by a graded module over the coordinate ring of the ambient projective space. The key difference between the category of sheaves and the category of modules is that a sheaf is not represented uniquely. However, there is a unique *maximal* graded module representing it, which is finitely generated (with certain provisos). For certain algorithms – computing cohomology, for example – any module representing the sheaf may be used. However for other calculations, such as explicit Riemann-Roch spaces or divisor maps, the full maximal module, containing the full space of global sections of the sheaf and its small Serre twists, is often required.

One of the basic operations, therefore, is the computation of the maximal module of a sheaf from its initial defining module. We have tried to do this efficiently in reasonable generality. The basic condition is that the support of the sheaf has irreducible components all of the same non-zero dimension. This will be described in more detail in the function descriptions that follow. The user does not have to explicitly make a call to perform the computation, but it may be carried out in the background and the result stored by several other functions.

A coherent sheaf $\mathcal{S}$ is defined by a graded module M over the polynomial ring $R = k[x_0, \ldots, x_n]$ and a subscheme X of $\mathbf{P}^n = Proj(R)$ on which M is supported. That is, the defining ideal $I \subseteq R$ of X annihilates M. In some contexts, X is unimportant and it doesn't matter whether $\mathcal{S}$ is thought of as a sheaf on X or on $\mathbf{P}^n$. In other cases, X plays a role: we can test whether $\mathcal{S}$ is locally free as a sheaf on X or take its dual. The sheaf $\mathcal{S}$ is just the coherent sheaf $\tilde{M}$ on X as described in Prop. 5.11, Section 5, Chapter II of [Har77], with M considered as a graded module over the homogeneous coordinate ring of X.

Sheaves are of type `ShfCoh`. There is also a type `ShfHom` for homomorphisms between sheaves supported on the same scheme X.

The algorithms used in the package are based on a number of computational commutative algebra tricks well-known to the experts.

113.2 Creation Functions

The general creation function for sheaves takes a graded module representing the sheaf and a scheme X on which it is supported. Special constructors are provided in the cases of the structure sheaf of X and the canonical sheaf of X, when X is locally Cohen-Macaulay and equidimensional. The user may also ask for Serre twists of a given sheaf. Other constructions deriving new sheaves from existing sheaves will be described in later sections.

```
Sheaf(M, X)
```

Given an ordinary projective scheme X and a module M over the coordinate ring of the ambient of X, such that M is annihilated by the defining ideal of X, this function returns the sheaf defined by graded module M on scheme X.

```
StructureSheaf(X)
StructureSheaf(X, n)
```

Given an ordinary projective scheme X, this function returns the structure sheaf $\mathcal{O}_X$ for X, which is the sheaf defined by the coordinate ring R_X of X, as a module. If the intrinsic is called with a second integer-valued argument n, the object returned is a twisted version of the sheaf, that is, Serre's twisting sheaf $\mathcal{O}_X(n)$, which has $R_X(n)$ as its associated graded module (see Section 5, Chapter II of [Har77]). These are all invertible sheaves on X and $\mathcal{O}_X(1)$ is the sheaf $\mathcal{O}_X(H)$ corresponding to the class of a hyperplane divisor H on X.

```
CanonicalSheaf(X)
CanonicalSheaf(X, n)
```

Given an ordinary projective scheme X, this function returns the canonical sheaf K_X for X. The scheme X should be an ordinary projective scheme which is equidimensional and locally Cohen-Macaulay. That is, all of the primary components of X should have the same dimension and its local rings should all be Cohen-Macaulay rings. These conditions aren't checked by Magma as the necessary computations can be very expensive in general. A non-singular variety always satisfies these conditions, and many singular normal varieties do also. For example, any curve or normal surface will be equidimensional and locally Cohen-Macaulay. The stronger condition of being *arithmetically* Cohen-Macaulay, can be checked by invoking the intrinsic `IsArithmeticallyCohenMacaulay` with the structure sheaf of X as argument.

Under these conditions, X has a canonical sheaf K_X, defined up to isomorphism, which acts as a dualising sheaf. See Section 7, Chapter III of [Har77] and Chapter

21 of [Eis95] for the module-theoretic background. For non-singular varieties, the canonical sheaf is the usual one: the highest alternating power of the sheaf of Kahler differentials. The function returns the canonical sheaf of X. It is computed from the dual complex to the minimal free resolution of the coordinate ring of X.

If the intrinsic is invoked with an additional integer argument n, it returns the nth Serre twist (see below) of the canonical sheaf $K_X(n)$. For a non-singular variety of dimension d, the map into projective space corresponding to $K_X(d-1)$ is the important *adjunction map*.

```
Twist(S, n)
```

Given a sheaf S, the function returns the nth Serre twist of $\mathcal{S}$, $\mathcal{S}(n) \cong \mathcal{S} \otimes_{\mathcal{O}_X} \mathcal{O}_X(n)$. If M is a module giving $\mathcal{S}$, then $M(n)$ gives $\mathcal{S}(n)$.

Example H113E1

We construct some sheaves associated with the smooth cubic surface defined by $x^3 + y^3 + z^3 + t^3$ in P^3.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> R := CoordinateRing(P);
> X := Scheme(P,x^3+y^3+z^3+t^3);
> OX := StructureSheaf(X);
```

We first examine the underlying graded module of the structure sheaf.

```
> Module(OX);
Reduced Module R^1/<relations>
Relations:
[x^3 + y^3 + z^3 + t^3]
```

Observe that the canonical sheaf KX of X is isomorphic to the twist $OX(-1)$ of the structure sheaf.

```
> KX := CanonicalSheaf(X);
> Module(KX);
Reduced Module R^1/<relations> with grading [1]
Relations:
[x^3 + y^3 + z^3 + t^3]
> Module(StructureSheaf(X,-1));
Reduced Module R^1/<relations> with grading [1]
Relations:
[x^3 + y^3 + z^3 + t^3]
```

Note that the module column weights are the negations of the Serre twist indices!

```
> Module(Twist(OX,-1));
Reduced Module R^1/<relations> with grading [1]
Relations (Groebner basis):
```

```
[x^3 + y^3 + z^3 + t^3]
```

The equations $x = z, y = t$ define an (exceptional) line in X. We can get its structure sheaf as a sheaf on X using the basic `Sheaf` constructor. The associated invertible sheaf $\mathcal{L}(Y)$ of Y as a divisor on X can be obtained from the `DivisorToSheaf` intrinsic described later in the chapter.

```
> IY := ideal<R|[x+z,y+t]>; // ideal of line
> OY := Sheaf(QuotientModule(IY),X);
> Module(OY);
Graded Module R^1/<relations>
Relations:
[x + z],
[y + t]
> Scheme(OY);
Scheme over Rational Field defined by
x^3 + y^3 + z^3 + t^3
```

`SheafOfDifferentials(X)`

`Maximize` BoolElt *Default :* `false`

Given an ordinary projective scheme X, this function returns the sheaf of 1-differentials on X, $\Omega^1_{X/k}$. The function computes the natural representing module for the sheaf coming from the embedding of X in projective space (see Section 8, Chapter II of [Har77]). If the parameter `Maximize` is `true`, then the maximal module representing this sheaf is computed and used to define it (see next section).

`TangentSheaf(X)`

`Maximize` BoolElt *Default :* `false`

For an ordinary projective scheme X, this function returns the sheaf of tangent vectors for X. The function computes the natural representing module for these sheaves coming from the embedding of X in projective space (see Section 8, Chapter II of [Har77]). If the parameter `Maximize` is `true`, then the maximal module representing this sheaf is computed and used to define it (see next section).

Combining either of the above intrinsics with the `IsLocallyFree` intrinsic, this gives an alternative method for checking non-singularity on varieties that are known to be (locally) Cohen-Macaulay. It is best to use the sheaf of differentials since that is generally easier to compute. This approach can be much faster for varieties having high codimension than the usual Jacobian method.

`HorrocksMumfordBundle(P)`

The projective space P should be ordinary projective 4-space $\mathbf{P}^4$ over a field. The function returns the locally free rank 2 sheaf on P which represents the Horrocks-Mumford bundle (see [HM73]). The scheme of vanishing of a general global section of this sheaf is a two dimensional Abelian variety in P.

113.3 Accessor Functions

The following functions provide an interface to conveniently extract the basic data from a coherent sheaf.

```
Module(S)
```

Returns the graded module that was used to define sheaf S.

```
Scheme(S)
```

Returns the ordinary projective scheme X on which the sheaf S is defined.

```
FullModule(S)
```

Computes and returns the maximal module M_{max} giving sheaf S. The nth graded piece of M_{max} is equal to the global sections of the Serre twist $S(n)$ as a finite dimensional vector space over k, the base field of the scheme X of S. Thus $M_{max} \cong \oplus_{n \in \mathbf{Z}} H^0(X, S(n))$ as in [Har77]. Here, it is implicitly assumed that the exact support of S on X has no irreducible components of dimension 0 and that there are no embedded associated prime places of dimension 0. More concretely, if M is a defining module for S with a possible non-zero finite torsion module for the redundant maximal ideal having been divided out, then no (homogeneous) associated prime of M has dimension 1. This assumption means that the terms in the above direct sum are 0 for $n \ll 0$ or equivalently that M_{max} is a finitely-generated module.

As mentioned in the introduction, a further assumption, which isn't checked, for the computation of M_{max} is that S is equidimensional, so that M actually has no embedded associated primes and the irreducible components of its exact support have the same non-zero dimension. It may be possible to avoid this assumption with more complex (and computationally heavy) code that works with an equidimensional decomposition of the defining module, but it suffices for many cases of interest (e.g., sheaves with trivial annihilator on a variety or equidimensional scheme).

The method used is basically the computation of the double dual of the defining module over an appropriate polynomial algebra A. A possible approach is to take A as the exact "supporting" algebra $k[x_0, \ldots, x_n]/I$ where the polynomial ring is the coordinate ring of the ambient of X and I is the exact annihilator of M. This would involve stronger assumptions on the support of S and the computation of the dualising module for this A. We choose instead to work with A as a *Noether normalisation* of the above A, which means that A is a simple polynomial ring and is its own dualising module (up to a shift in grading). Then M is re-expressed as a module over this A, M_{max} is computed as a module over A and finally is recovered as a module over $k[x_0, \ldots, x_n]$ by keeping track of the multiplication maps by the x_i variables which don't occur in A.

The module M_{max} is stored so that it is only computed once.

`GlobalSectionSubmodule(S)`

Given a sheaf S, this function returns the submodule of the maximal module M_{max} generated in degrees ≥ 0, that is $\oplus_{n\geq 0} H^0(X, S(n))$.

`SaturateSheaf(~S)`

Procedure to compute and store (but not return) the maximal module M_{max} of the sheaf S.

Example H113E2

The classic example of a natural module which is unsaturated (non-maximal) defining a sheaf is the coordinate ring R of a non-projectively normal non-singular projective variety X. The ring R defines the structure sheaf as usual, but not maximally. By definition, R isn't integrally closed. Its integral closure R_1 is an extension ring, inheriting its natural grading and agreeing with R in all but finitely many graded parts. In fact, R_1 considered as an R-module is precisely the maximal graded module of the structure sheaf!

Such a situation can very commonly arise when a non-singular variety is projected down isomorphically into a subspace of its ambient projective space. The projected down image X is then not even linearly-normal: the degree one graded part of its coordinate ring is missing coordinates that were eliminated in the projection. These must reoccur in the graded R-module that is computed as the maximal module of the structure sheaf.

In the following example, X is taken as the non-singular projection into P^3 of a degree 4 rational normal curve (which naturally lives in P^4). We can see the difference between the maximal module of the structure sheaf and the coordinate ring using Hilbert series. In fact, they just differ by dimension 1 in the 1-graded part, corresponding to that missing coordinate!

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Scheme(P3,[
> y^3 - y*z^2 - 2*y^2*t - 2*x*z*t - 3*y*z*t + z^2*t - y*t^2 + 2*z*t^2 + 2*t^3,
> x^2*z + x*z^2 + y*z^2 + 3*x*z*t + 2*y*z*t - z^2*t + y*t^2 - 2*z*t^2 - 2*t^3,
> y^2*z - y*z^2 + y^2*t - x*z*t - 4*y*z*t + z^2*t - 3*y*t^2 + 2*z*t^2 + 2*t^3,
> x*y - x*z - x*t + y*t]);
> OX := StructureSheaf(X);
> M1 := Module(OX);
> M2 := FullModule(OX);
> h1 := HilbertSeries(M1); h1;
(-t^3 + 2*t^2 + 2*t + 1)/(t^2 - 2*t + 1)
> h2 := HilbertSeries(M2); h2;
(3*t + 1)/(t^2 - 2*t + 1)
> h2-h1;
t
```

113.4 Basic Constructions

The following functions give some basic constructions on sheaves.

`TensorProduct(S, T)`

`Maximize` BoolElt *Default :* `false`

`TensorPower(S, n)`

`Maximize` BoolElt *Default :* `true`

The first intrinsic gives the tensor product (over $\mathcal{O}_X$) of two sheaves on the same scheme X. The second gives the nth tensor power of S if $n > 0$, the $(-n)$th tensor power of the dual (see below) of S if $n < 0$ and the structure sheaf $\mathcal{O}_X$ if $n = 0$.

Defining modules for these constructions are taken as the appropriate tensor products of modules for the constituent sheaves when the parameter `Maximize` is `false`. The user should note that this is the archetypal case where the module constructed to define the resulting sheaf can be far from maximal, even when the defining modules of S and T are maximal. The rank of the presentation of the tensor power of a module rises rapidly with n. Thus, it is usually a good idea to set `Maximize` to `true`, which means that the maximal module of the result is computed and also used as its defining module.

`Dual(S)`

For the sheaf S on a scheme X, the function returns the dual sheaf $\mathrm{Hom}_{\mathcal{O}_X}(S, \mathcal{O}_X)$.

`SheafHoms(S, T)`

For S and T sheaves on the same scheme X, the function returns the sheaf $H = \mathrm{Hom}_{\mathcal{O}_X}(S, T)$. The module defining H is $\mathrm{Hom}(M_{\max}, N_{\max})$, where $M_{\max}$ and $N_{\max}$ are the maximal modules of S and T. This module, M_H, is the maximal module of H.

Also returned is a map that takes a homogeneous element of M_H (which can be recovered with `Module(H)` or `FullModule(H)`) of degree d to the sheaf homomorphism of degree d that it represents (see the next section for information about sheaf homomorphisms). All sheaf homomorphisms can be obtained this way.

`DirectSum(S, T)`

For S and T sheaves on the same scheme X, this function returns the sheaf direct sum $S \oplus T$.

`Restriction(S, Y)`

`Check` BoolElt *Default :* `true`

Given a sheaf S on a scheme X and a subscheme Y of X, the function returns the restriction of S to Y. A check that Y is a subscheme of X will be performed only if the parameter `Check` is `true` (the default).

Example H113E3

We look at the well-known example of a ruling L on a (singular) projective quadric cone X in P^3. We find the associated invertible sheaf $O_X(L)$ using the `DivisorToSheaf` intrinsic. The tensor square of this sheaf is $O_X(2L)$ which is just isomorphic to the $O_X(1)$ Serre twist of the structure sheaf, as $2L$ is a hyperplane section. We verify this by getting the tensor and inspection. Of course we need to saturate the result, illustrating that the basic tensor power of maximal modules usually does not result in a maximal module.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> R := CoordinateRing(P);
> X := Scheme(P,x*y-z^2); // singular projective quadric
> IL := ideal<R|z,y>;     // line y=z=0 on X
> OL := DivisorToSheaf(X,IL); // associated sheaf O(L)
```

We first make sure that OL is saturated.

```
> SaturateSheaf(~OL);
> Module(OL);
Graded Module R^2/<relations>
Relations:
[ y, -z],
[ z, -x]
> O2L := TensorProduct(OL,OL); // or TensorPower(OL,2)
> Module(O2L);
Graded Module R^4/<relations>
Relations:
[ y,  0, -z,  0],
[ 0,  y,  0, -z],
[ z,  0, -x,  0],
[ 0,  z,  0, -x],
[ y, -z,  0,  0],
[ z, -x,  0,  0],
[ 0,  0,  y, -z],
[ 0,  0,  z, -x]
```

Finally, we get the maximum module – just that of OX(1)!

```
> FullModule(O2L);
Reduced Module R^1/<relations> with grading [-1]
Relations:
[x*y - z^2]
```

113.5 Sheaf Homomorphisms

This section describes some basic functionality for homomorphisms between sheaves defined on the same scheme. A sheaf homomorphism is represented by a module homomorphism between representing modules (defining, maximal or global section modules) for the two sheaves. Strictly, only (degree 0) homomorphisms that preserve the module gradings should be allowed but, for flexibility, we allow "homogeneous" homomorphisms that uniformly shift the grading by d and should be thought of as sheaf homomorphisms from the domain sheaf to the dth Serre twist of the codomain sheaf. We then say that the homomorphism is of degree d as for module homomorphisms.

There are some basic constructors and accessor functions, a function to "expand" a chain of homomorphisms to a single homomorphism and image, kernel and cokernel functions. The type of a sheaf homomorphism is `ShfHom`. Note that this is NOT a `Map` subtype, so that a sheaf homomorphism doesn't automatically inherit all of the usual map properties.

`SheafHomomorphism(S, T, h)`

Given sheaves S and T on the same scheme X and a module homomorphism h between M_0 and N_0, this function returns a sheaf homomorphism from S to T. Here M_0 is one of the defining, maximal or global section modules of S and N_0 is a similar module for T. The homomorphism h must be a homogeneous module homomorphism as returned by `IsHomogeneous` and if d is its degree then the homomorphism returned is really one from S to $T(d)$ in the category of $\mathcal{O}_X$-sheaves.

For the construction of sheaf homomorphisms see also `SheafHoms`.

`Domain(f)`

The domain of the sheaf homomorphism f.

`Codomain(f)`

The codomain of the sheaf homomorphism f.

`Degree(f)`

The degree of the sheaf homomorphism f as defined in the introduction to this section.

`ModuleHomomorphism(f)`

The underlying homogeneous graded module homomorphism of the sheaf homomorphism f.

`Kernel(f)`

Given a sheaf homomorphism f, this function returns the kernel of f and its inclusion homomorphism into the domain of f.

```
Image(f)
```

Given a sheaf f, this function returns the image, I, of f and two sheaf homomorphisms g and h. If f has degree d and S, T are its domain and codomain, then I is a subsheaf of $T(d)$. The second return value g is the restriction of f from S to I and the third return value h is the inclusion of I in $T(d)$, so that g also has degree d and h has degree 0.

```
Cokernel(f)
```

Given a sheaf homomorphism f, this function returns the cokernel of f and also the quotient homomorphism from the codomain to it.

Strictly speaking, if f has degree d and S, T are its domain and codomain, here we are taking f to be a homomorphism from $S(d) \leftarrow T$ rather than from $S \leftarrow T(d)$.

```
Expand(hms)
```

If $hms = [h_1, \dots, h_n]$ is a sequence of sheaf homomorphisms, then this function returns the composition of homomorphisms $h_1 * h_2 * \dots h_n$. The domain of h_2 must be the codomain of h_1 etc. and the stronger condition that the underlying module homomorphisms must be composable also holds. So the domain of `ModuleHomomorphism(h2)` must be the codomain of `ModuleHomomorphism(h1)` etc.

113.6 Divisor Maps and Riemann-Roch Spaces

As stated at the beginning of the chapter, one of our main initial aims in introducing sheaf machinery has been to provide a way of computing the (rational) maps associated to invertible sheaves in reasonable generality (see Section 7, Chapter 2 of [Har77]) and similarly for effective Cartier divisors (as closed subschemes) in the form of the map or their Riemann-Roch spaces. This section describes the main intrinsics. We hope to add further functionality to capture the correspondence between divisors and invertible sheaves in future releases.

```
DivisorMap(S)
```

`graphmap` BoolElt *Default :* `false`

Given an invertible sheaf S on the scheme X this function returns the rational map from X into the projective space associated to S. For efficiency, the invertibility of S is not checked, so that if the user is unsure whether a potential S actually is invertible (ie, locally free of rank one) he should apply the `IsLocallyFree` intrinsic.

The rational map that is returned can be thought of as $X \to Proj(R) \to \mathbf{P}^r$ where R is the graded k-subalgebra of the graded ring $\oplus_{n\geq 0} H^0(X, S^{\otimes n})$ generated by the weight 1 subspace $H^0(X, S)$ – the space of global sections of S – and the map to $\mathbf{P}^r$, where $r+1$ is the dimension of the space of global sections, is that induced by choosing a basis for the global sections. The divisor map, as usual, is only unique up to a linear change of coordinates of the codomain. The map is defined on the open subscheme of X where S is generated by global sections. Also returned is the image of the map on X.

In most cases, the map returned is a graph map of type `MapSchGrph` (see Section 112.14.7). This computation naturally computes the graph of the map and, in complicated situations, it is not particularly efficient to convert this to the more usual `MapSch` which will be defined by very nasty, high degree polynomials (often with a large base scheme) without some further specialised reduction routine. The user can however convert to a `MapSch` using `SchemeGraphMapToSchemeMap`. In cases when the sheaf has been constructed from a divisor using the `DivisorToSheaf` intrinsic below with the `GetMax` parameter `true`, so that a Riemann-Roch space has been stored, a traditional `MapSch` is returned. If in doubt, the user can distinguish using the `Type` intrinsic. Sometimes the user may still want a `MapSchGrph` in the latter case (e.g. because it is maximally defined, it is good for getting the genuine inverse image of a point/subscheme without components of the base scheme which appear for a non-maximally defined `MapSch`). This can be forced by setting the parameter `graphmap` to `true`.

The major stage of the computation is the determination of the graph of the map. An ideal defining the graph can be written down directly from the relation matrix of a minimal presentation of the global section submodule M_0 of S, and this ideal then only needs to be saturated with respect to an appropriate domain variable. The submodule M_0 is computed (and stored) as described earlier in the chapter.

```
DivisorToSheaf(X, I)
```

`GetMax` BoolElt *Default :* `true`

```
RiemannRochBasis(X, I)
```

Given an ordinary, projective scheme X and an ideal I of the coordinate ring of the ambient of X that defines a subscheme D of X that is an effective Cartier (locally principal) divisor of X, this function returns the invertible sheaf corresponding to the divisor class of D, commonly denoted $\mathcal{L}(D)$ (see Section 6, Chapter 2 of [Har77]). The conditions require D to be purely of codimension 1 in X and that it is everywhere locally defined by a single equation. Again for efficiency, Magma does not perform the computationally expensive checks to verify that D is locally principal within X. If X is a non-singular variety, then a closed subscheme of codimension 1 is automatically Cartier.

If `GetMax` is `true`, then the maximal module of $\mathcal{L}(D)$ is computed and an explicit basis for the Riemann-Roch space $L(D)$ is computed and stored along the way. This basis is of the form $[G_1/G, \ldots, G_n/G]$, where G and the G_i are homogeneous polynomials of some degree d on the ambient of X and the G_i/G are the usual rational functions restricted to X.

If instead of `DivisorToSheaf`, the intrinsic `RiemannRochBasis` is called, then the above procedure is carried out and a basis of the Riemann Roch space is returned as the sequence of numerators $[G_1, \ldots, G_n]$ and the denominator G, along with the sheaf $\mathcal{L}(D)$. If $\mathcal{L}(D)$ has been computed from `DivisorToSheaf` and returned as S, then the RiemannRoch basis can be recovered at a later stage from the *attribute* of S, `rr_space`. This attribute, if assigned, contains a pair consisting of the above sequence of numerators and the denominator.

The algorithm used is based on the following observation. If we choose $r > 0$ such that I contains a homogeneous polynomial G of degree r that doesn't lie in the ideal of X, I_X (which is a proper subideal of I), then there is a "complementary" divisor E of X such that $rH \sim D + E$, where H is a hyperplane divisor of X. Then $\mathcal{L}(D) \simeq \mathcal{L}(-E)(r)$ and $\mathcal{L}(-E)$ is represented by the module I_E/I_X, where I_E is the ideal of E, a subscheme of X (see Prop 6.18 of the above reference). Once a suitable G is found, I_E is computed by invoking intrinsics `ColonIdeal` and `Saturation` a few times. If the `GetMax` option is on, r is chosen large enough so that $H^1(I_X(m)), m \geq r$ vanishes, which guarantees that we end up with a maximal representing module and can get a full basis of Riemann-Roch numerators with G as the denominator.

Example H113E4

As a simple example, we consider a degree 3 rational scroll in $\mathbf{P}^4$. This is a ruled surface that contains a family of disjoint lines. If l is a line in the family, then the divisor map for $\mathcal{L}(l)$ is a map to the projective line, the fibres of which are the lines of the family. We take such a scroll X and line l and get the Riemann-Roch space $L(l)$ and the divisor map down to $\mathbf{P}^1$.

```
> P4<a,b,c,d,e> := ProjectiveSpace(Rationals(),4);
> X := Scheme(P4,[a*b - c^2, a*d - c*e, c*d - b*e]);
> Il := ideal<CoordinateRing(P4)|[a,c,e]>; // ideal of l
> rr_seq,G, Sl := RiemannRochBasis(X,Il);
> rr_seq; G;
[
    d,
    e
]
e
```

Thus, 1 and d/e are a basis for the rational functions in $L(l)$.

```
> fib_mp := DivisorMap(Sl);
> fib_mp;
Mapping from: Sch: X to Projective Space of dimension 1
Variables: $.1, $.2
with equations :
d
e
```

Here the divisor map is not a graph map and is not maximally defined. So we extend it to make it so. Note that the fibres are lines.

```
> fib_mp := Extend(fib_mp);
> (Codomain(fib_mp)![1,0])@@fib_mp;
Scheme over Rational Field defined by
a,
c,
e,
a*b - c^2,
a*d - c*e,
c*d - b*e
> (Codomain(fib_mp)![0,1])@@fib_mp;
c,
b,
d,
a*b - c^2,
a*d - c*e,
c*d - b*e
```

Alternatively, we could ask for fib_mp as a MapSchGrph and not have to extend it.

```
> fib_mp := DivisorMap(S1 : graphmap := true);
> Type(fib_mp);
MapSchGrph
> (Codomain(fib_mp)![1,0])@@fib_mp;
Scheme over Rational Field defined by
a,
c,
e,
a*b - c^2,
a*d - c*e,
c*d - b*e
```

Example H113E5

As a second example, we consider the degree 3 Del Pezzo surface example from the Del Pezzo chapter. There we mapped it to a degree 6 Del Pezzo surface by blowing down 3 disjoint lines in an explicit fashion. We do the same thing here using the sheaf machinery.

First we get the surface X_3 and the union of the 3 lines L_{123}:-

```
> R3<x,y,z,t> := PolynomialRing(Rationals(),4,"grevlex");
> P3 := Proj(R3);
```

We set up the equation defining the degree 3 surface:

```
> F := -x^2*z + x*z^2 - y*z^2 + x^2*t - y^2*t - y*z*t + x*t^2 + y*t^2;
> X3 := Scheme(P3,F);
```

Get the ideal defining the union of the 3 lines:

```
> I1 := ideal<R3|[x,y]>; // line 1 L1
> I2 := ideal<R3|[z,t]>; // line 2 L2
> I3 := ideal<R3|[x-z,y-t]>; //line 3 L3
> I := I1*I2*I3; // (non-saturated) ideal of L1+L2+L3 = L123
```

Now we blow down to get the degree 6 Del Pezzo in $\mathbf{P}^6$. The divisor we need for the map is $H + L_{123}$ where H is a hyperplane section. We get this simply by twisting the sheaf corresponding to L_{123} once.

```
> S123 := DivisorToSheaf(X3,I);
> H6 := Twist(S123,1); // sheaf of H+L123
> mp, X := DivisorMap(H6);
> X;
Scheme over Rational Field defined by
y[1]*y[2] - y[2]*y[3] - y[4]*y[5] + y[1]*y[6] + 3*y[2]*y[6] - y[4]*y[6] + y[6]^2
    + y[1]*y[7] + 2*y[2]*y[7] - y[4]*y[7] - y[5]*y[7] + 3*y[6]*y[7],
y[2]^2 - y[2]*y[3] + 2*y[2]*y[6] + y[6]^2 + 2*y[2]*y[7] + 3*y[6]*y[7],
y[1]*y[3] - y[2]*y[3] + 2*y[2]*y[6] - y[5]*y[6] + y[6]^2 + y[2]*y[7] - y[4]*y[7]
    + 3*y[6]*y[7],
y[2]*y[4] - y[2]*y[6] + y[4]*y[6] + y[4]*y[7] + y[5]*y[7],
y[3]*y[4] - y[2]*y[6] - y[6]^2 - y[6]*y[7] + y[7]^2,
y[4]^2 - y[4]*y[6] + y[5]*y[6] + y[1]*y[7] - y[2]*y[7] + y[4]*y[7] + y[5]*y[7],
y[2]*y[5] - y[4]*y[6] + y[5]*y[6] - y[2]*y[7] + y[4]*y[7] + y[5]*y[7],
y[3]*y[5] - y[6]^2 - y[2]*y[7] - y[7]^2,
y[5]^2 - y[1]*y[6] + y[2]*y[6] - 2*y[4]*y[6] + y[5]*y[6] + y[1]*y[7] - y[2]*y[7]
> Dimension(X); Degree(X);
2
6
```

113.7 Predicates

This subsection describes tests for several important properties of coherent sheaves. It contains an isomorphism test that, combined with DivisorToSheaf, can be used as a test for linear equivalence of Cartier divisors.

IsLocallyFree(S)

UseFitting BoolElt *Default :* true

Given a sheaf S on ordinary projective scheme X, this function returns true if and only if S is a locally free sheaf on X of constant rank and, if so, also returns its rank.

Our original implementation was very fast but unfortunately incorrect! We have modified it to the algorithm described below which uses an “étale stratification” of X.

The more straightforward method uses Fitting ideals of the module M (or M_{max}) of S. If the possible rank is d (as determined from Hilbert polynomials), it is required to check that the saturation of the dth Fitting ideal is the full ring and that the $(d-1)$th lies in the saturated ideal of X. This is now the default method, but can be extremely slow and use a large amount of memory. Our alternative method is much slower than it was but we still find that it can be much faster than the Fitting ideal method for a low dimensional X in a high dimensional ambient and a sheaf S whose (maximal) module has a presentation with a reasonably large minimal number of both generators and relations.

To use the alternative method, the user can set the `UseFitting` parameter to `false`. For this method, it is assumed that X is equidimensional (all of its primary components have the same dimension), (locally) Cohen-Macaulay and connected. Magma does not check these conditions. The alternative method is described below.

The equidimensional and locally Cohen-Macaulay assumptions imply that X is faithfully flat over $P_0 = Proj(R_0)$ for a Noether normalisation R_0 of the coordinate ring of X. Standard flatness properties mean that S being locally free over X implies that it is locally free as a sheaf over P_0, which is just a full projective space. Serre's criterion (see [Ser55]) states that the latter is true if and only if all intermediate cohomology rings $H^i(P_0, S(q))$ vanish for $q \ll 0$. If M_{max} is the maximal graded module of S, this translates to all intermediate $Ext(M_{max}, R_0)$ R_0-modules being finite length, which in turn translates to the dual complex to the minimal free resolution of M_{max} as an R_0-module, having finite-length homology groups at all intermediate places. Rather than actually computing homology modules, we can further translate this condition into a number of equality tests for Hilbert polynomials of cokernels of the maps between free modules in the dual complex. This operation seems to be fairly fast and efficient in practise.

The above gives a necessary condition for local freeness over X (and gives the rank) but it is only sufficient over the Zariski open subset of X over which $X \to P_0$ is unramified. We have adapted it to be applied inductively over a chain of closed subschemes of X, which is what we refer to as the étale stratification of X. At each level, we have a subscheme Y of X and a possibly empty collection of polynomials $\{F_i\}$ which are non-zero divisors on Y and such that a chosen Noether normalisation of Y is unramified (equivalently, étale) outside of the subschemes Y_i defined by F_i adjoined to the equations of Y. These subschemes lie at the next level down. The conditions imply that all of the Y_i are equidimensional and Cohen-Macaulay. For Y_i of positive dimension, we apply the above test to the restriction of S to Y_i, checking that the rank is the same as the rank determined for X at the top level if the test is passed and also that F_i is not a zero-divisor on the module of the restricted sheaf. For Y_i of dimension 0, we perform a more direct test.

The overall algorithm generally takes much longer than just applying the main test to X. The étale stratification can take some time to compute when X lies in higher dimensional ambients. However, the main problem is that the degrees of the Y_i increase as we go down the chain (the subscheme defined by F_i has the degree of Y_i multiplied by the degree of F_i as its degree) and in practice (mainly working with surfaces X), the most time is taken in the bottom level checks on high degree zero-dimensional subschemes. Once computed, the étale stratification is stored with X, so does not need to be recomputed for tests on other sheaves. However, it relies on finding Noether normalisations at each level that are generically unramified, i.e. separable. Currently we do not check the condition and it can fail in small positive characteristic leading to a crash or wrong results.

```
IsIsomorphic(S, T)
```

```
IsIsomorphicWithTwist(S, T)
```

For S and T coherent sheaves on the same base scheme X, this function returns `true` if and only if S is isomorphic to T or (for the second intrinsic) to a Serre twist $T(d)$ of T. In either case, an isomorphism is returned, if one exists and for the second intrinsic, the twist d is also returned as the second return value (so the isomorphism is between S and $T(d)$).

For the implementation, we first do a quick Hilbert polynomial check and then a Betti number check for the maximal modules M_{max} and N_{max} of S and T. This gives necessary conditions for an isomorphism and the possible d in the "with twist" case. Then we look for an isomorphism in the finite dimensional space of homomorphisms between M_{max} and N_{max}. We could have chosen to work with the homomorphisms between the truncated modules with gradings greater than or equal to N, for some N greater than or equal to the regularity of any defining modules for the two sheaves, but these have much larger presentations in general so the computation of homomorphisms is slower.

To look for isomorphisms, we look at the "zero degree" subblocks of the matrices giving a basis to the space of all homomorphisms. This reduces the problem to determining whether there is an invertible matrix in a space of $n \times n$ matrices over the base field. This is known to be a difficult problem in general and currently our implementation is rather weak at this point. We hope to improve it for future releases.

```
IsArithmeticallyCohenMacaulay(S)
```

Given a sheaf S on an ordinary projective scheme X, this function returns `true` if and only if the maximal graded module M_{max} of S is a Cohen-Macaulay module over the coordinate ring of X.

A scheme X is called arithmetically Cohen-Macaulay if and only if its coordinate ring is a Cohen-Macaulay ring. This is then true if and only if its coordinate ring is equal to the maximal module of $\mathcal{O}_X$ and the intrinsic returns `true`for $\mathcal{O}_X$.

This is a fairly straightforward computation once M_{max} has been determined. If we already know the structure of M_{max} as a module over a Noether normalisation of the coordinate ring of X, it is an immediate freeness check. Otherwise it is a straightforward depth calculation from a minimal free resolution of M_{max} as a graded module over the coordinate ring of the ambient of X.

113.8 Miscellaneous

```
CohomologyDimension(S, r, n)
```

Given a sheaf S and integers r and n, this function returns the dimension over the base field of the r-th cohomology group of the n-th Serre twist of S, $H^r(X, S(n)$.

This just calls the equivalent function for the maximal module of S or its defining module, if the maximal module has not yet been computed. Note that, in practice, it may often be much faster to use the maximal module so it may be desirable to call `SaturateSheaf` before doing any cohomology computations.

```
DimensionOfGlobalSections(S)
```

This returns the same dimension as `CohomologyDimension(S,0,0)` – the dimension of the space of global sections of S – but it is computed in a different way that is usually faster. It uses some straightforward linear algebra to compute the dimension of the zero-th graded part of the maximal module of S given as a presentation module.

```
IntersectionPairing(S, T)
```

If S and T are invertible sheaves on a nonsingular surface X, representing divisor classes D and E, this function returns the surface intersection number $D.E$.

Only minimal checks are made on the validity of the input data. The computation is a standard one using the Hilbert polynomials of S, T and their tensor product.

```
ZeroSubscheme(S, s)
```

The sheaf S should be a locally free sheaf on a scheme X (local freeness is *not* checked). The element s should be a homogeneous element of the defining, maximal or global section modules of S. Then s represents a global section of the twisted sheaf $S(d)$ if s is homogeneous of degree d. The intrinsic returns the vanishing subscheme of s: the largest subscheme of X on which s restricts to a zero section. If S is invertible of the form $\mathcal{L}(D)$, for example, s represents an effective divisor D_s in the linear system $|D + dH|$ (if it is non-zero) and the vanishing subscheme is D_s as a subscheme of X. Locally, for a Zariski-open set U over which there is an isomorphism $S(d)_U \cong {\mathcal{O}_X}_U^n$, $s|_U$ corresponds to an n-tuple of functions $(f_1, \ldots, f_n)$ on U and the vanishing subscheme restricted to U is the closed subscheme of U defined by the ideal $\langle f_1, \ldots, f_n \rangle$.

113.9 Examples

In this section we present some extended examples illustrating various features of the sheaf machinery.

Example H113E6

In this example, we consider a surface X from a special family of rational surfaces of degree 10 in $\mathbf{P}^4$. This family is described by Decker, Ein and Schreyer in Section 2.1 of [DES93]. They have sectional genus 9 and are isomorphic to the plane blown up in 18 points in special position which give 18 exceptional curves in X. The embedding into $\mathbf{P}^4$ is such that four of these exceptional curves are of degree 3, seven curves are of degree 2 and seven curves are of degree 1.

The adjunction map on X is the map corresponding to the divisor $K_X + H$, where K_X is a canonical divisor and H is a hyperplane section, or, equivalently to the sheaf $\mathcal{K}_X(1)$ where $\mathcal{K}_X$ is the canonical sheaf. In our example, the adjunction map maps X to a smooth surface X_1 of degree 13 in $\mathbf{P}^8$ blowing down the seven degree 1 exceptional curves to points and reducing the degrees of the others by 1. The adjunction map on X_1 blows down the seven exceptional curves originally of degree 2 to points and maps X_2 to an anticanonically embedded degree 5 Del Pezzo surface in $\mathbf{P}^5$.

We take a randomly generated surface from this family over a small finite field ($\mathbf{F}_{17}$) and illustrate this process by explicitly computing the adjunction maps and images X_1 and X_2. We show that the intersection pairings of the canonical divisor and hyperplane sections on X, X_1, X_2 are as expected and that X_2 really is an anticanonically embedded Del Pezzo surface. We also expand the composition of the two divisor maps and show that the resulting map is indeed a birational map from X onto X_2.

These surfaces X are defined by one degree 4 and ten degree 5 polynomials in $\mathbf{P}^4$. The embedding is quite a complex one and it is hard to construct one with defining polynomials which are at all sparse. This makes it fairly challenging for explicit computation and also means that an example takes up a lot of page space! An example with relatively small coefficients over $\mathbf{Q}$ can also be processed, though the total running time is a few minutes. Also, the resulting X_2 tends to have very large coefficients. Here we get no coefficient blow-up and X_2 is a much simpler looking surface than X.

```
> P<[x]> := ProjectiveSpace(GF(17),4);
> X := Scheme(P, [
>    10*x[1]^4 + 13*x[1]^3*x[2] + 8*x[1]*x[2]^3 + 4*x[2]^4 + 6*x[1]^3*x[3] +
>        15*x[1]^2*x[2]*x[3] + 14*x[2]^3*x[3] + x[1]^2*x[3]^2 +
>        13*x[1]*x[2]*x[3]^2 + 3*x[2]^2*x[3]^2 + 9*x[1]*x[3]^3 + 2*x[2]*x[3]^3 +
>        10*x[3]^4 + 15*x[1]^3*x[4] + 4*x[1]^2*x[2]*x[4] + 3*x[1]*x[2]^2*x[4] +
>        7*x[2]^3*x[4] + 9*x[1]^2*x[3]*x[4] + 3*x[1]*x[2]*x[3]*x[4] +
>        9*x[2]^2*x[3]*x[4] + 11*x[1]*x[3]^2*x[4] + 6*x[2]*x[3]^2*x[4] +
>        15*x[3]^3*x[4] + x[1]^2*x[4]^2 + 4*x[1]*x[2]*x[4]^2 + 2*x[2]^2*x[4]^2 +
>        12*x[1]*x[3]*x[4]^2 + 8*x[2]*x[3]*x[4]^2 + 9*x[3]^2*x[4]^2 +
>        10*x[1]*x[4]^3 + 5*x[2]*x[4]^3 + 14*x[3]*x[4]^3 + 4*x[1]^3*x[5] +
>        16*x[1]^2*x[2]*x[5] + 15*x[2]^3*x[5] + 13*x[1]^2*x[3]*x[5] +
>        13*x[1]*x[2]*x[3]*x[5] + 10*x[2]^2*x[3]*x[5] + 15*x[1]*x[3]^2*x[5] +
>        7*x[2]*x[3]^2*x[5] + 14*x[3]^3*x[5] + 11*x[1]^2*x[4]*x[5] +
>        10*x[1]*x[2]*x[4]*x[5] + 4*x[2]^2*x[4]*x[5] + x[1]*x[3]*x[4]*x[5] +
>        12*x[2]*x[3]*x[4]*x[5] + 8*x[3]^2*x[4]*x[5] + 5*x[1]*x[4]^2*x[5] +
>        5*x[2]*x[4]^2*x[5] + 11*x[3]*x[4]^2*x[5] + 10*x[4]^3*x[5] +
>        12*x[1]^2*x[5]^2 + 8*x[1]*x[2]*x[5]^2 + 16*x[2]^2*x[5]^2 +
>        12*x[1]*x[3]*x[5]^2 + x[2]*x[3]*x[5]^2 + 14*x[3]^2*x[5]^2 +
>        8*x[1]*x[4]*x[5]^2 + x[2]*x[4]*x[5]^2 + 3*x[3]*x[4]*x[5]^2 +
>        5*x[4]^2*x[5]^2 + 11*x[1]*x[5]^3 + 13*x[2]*x[5]^3 + 5*x[3]*x[5]^3 +
>        9*x[4]*x[5]^3 + 8*x[5]^4,
>    9*x[1]^4*x[4] + 14*x[1]^3*x[2]*x[4] + 5*x[1]^2*x[2]^2*x[4] +
>        2*x[1]*x[2]^3*x[4] + 2*x[1]^3*x[3]*x[4] + 7*x[1]^2*x[2]*x[3]*x[4] +
>        5*x[1]*x[2]^2*x[3]*x[4] + 7*x[2]^3*x[3]*x[4] + 9*x[1]^2*x[3]^2*x[4] +
>        12*x[1]*x[2]*x[3]^2*x[4] + 2*x[2]^2*x[3]^2*x[4] + 9*x[1]*x[3]^3*x[4] +
>        2*x[2]*x[3]^3*x[4] + x[3]^4*x[4] + 3*x[1]^3*x[4]^2 +
>        5*x[1]^2*x[2]*x[4]^2 + 7*x[1]*x[2]^2*x[4]^2 + 13*x[2]^3*x[4]^2 +
>        11*x[1]^2*x[3]*x[4]^2 + 4*x[1]*x[2]*x[3]*x[4]^2 + 11*x[2]^2*x[3]*x[4]^2
>        + 14*x[1]*x[3]^2*x[4]^2 + 16*x[2]*x[3]^2*x[4]^2 + 15*x[1]^2*x[4]^3 +
>        11*x[1]*x[2]*x[4]^3 + 5*x[2]^2*x[4]^3 + 6*x[1]*x[3]*x[4]^3 +
>        9*x[2]*x[3]*x[4]^3 + 16*x[3]^2*x[4]^3 + 9*x[2]*x[4]^4 + 15*x[3]*x[4]^4 +
>        14*x[4]^5 + 2*x[1]^3*x[2]*x[5] + 6*x[1]^2*x[2]^2*x[5] +
>        3*x[1]*x[2]^3*x[5] + 16*x[2]^4*x[5] + 15*x[1]^3*x[3]*x[5] +
>        6*x[1]*x[2]^2*x[3]*x[5] + 10*x[2]^3*x[3]*x[5] + 14*x[1]^2*x[3]^2*x[5] +
>        13*x[1]*x[2]*x[3]^2*x[5] + 4*x[2]^2*x[3]^2*x[5] + 16*x[1]*x[3]^3*x[5] +
>        13*x[3]^4*x[5] + 14*x[1]^3*x[4]*x[5] + 9*x[1]^2*x[2]*x[4]*x[5] +
>        16*x[1]*x[2]^2*x[4]*x[5] + 14*x[2]^3*x[4]*x[5] +
>        6*x[1]*x[2]*x[3]*x[4]*x[5] + 6*x[2]^2*x[3]*x[4]*x[5] +
>        3*x[1]*x[3]^2*x[4]*x[5] + 7*x[2]*x[3]^2*x[4]*x[5] + 7*x[3]^3*x[4]*x[5] +
>        2*x[1]^2*x[4]^2*x[5] + 15*x[1]*x[2]*x[4]^2*x[5] +
>        9*x[1]*x[3]*x[4]^2*x[5] + 14*x[3]^2*x[4]^2*x[5] + 14*x[1]*x[4]^3*x[5] +
>        6*x[2]*x[4]^3*x[5] + 12*x[3]*x[4]^3*x[5] + 3*x[4]^4*x[5] +
>        9*x[1]^3*x[5]^2 + 12*x[1]^2*x[2]*x[5]^2 + 16*x[1]*x[2]^2*x[5]^2 +
>        x[2]^3*x[5]^2 + 7*x[1]^2*x[3]*x[5]^2 + 5*x[1]*x[2]*x[3]*x[5]^2 +
>        8*x[2]^2*x[3]*x[5]^2 + 2*x[1]*x[3]^2*x[5]^2 + 4*x[2]*x[3]^2*x[5]^2 +
>        13*x[3]^3*x[5]^2 + 7*x[1]^2*x[4]*x[5]^2 + 6*x[2]^2*x[4]*x[5]^2 +
>        16*x[1]*x[3]*x[4]*x[5]^2 + 15*x[2]*x[3]*x[4]*x[5]^2 +
>        7*x[3]^2*x[4]*x[5]^2 + 6*x[1]*x[4]^2*x[5]^2 + 3*x[2]*x[4]^2*x[5]^2 +
>        16*x[3]*x[4]^2*x[5]^2 + 15*x[4]^3*x[5]^2 + x[1]^2*x[5]^3 +
>        13*x[1]*x[2]*x[5]^3 + 6*x[2]^2*x[5]^3 + 8*x[1]*x[3]*x[5]^3 +
>        x[2]*x[3]*x[5]^3 + 9*x[3]^2*x[5]^3 + 3*x[1]*x[4]*x[5]^3 +
>        14*x[2]*x[4]*x[5]^3 + 8*x[3]*x[4]*x[5]^3 + 14*x[4]^2*x[5]^3 +
>        16*x[1]*x[5]^4 + 2*x[2]*x[5]^4 + 7*x[3]*x[5]^4 + 7*x[4]*x[5]^4 +
>        11*x[5]^5,
>    13*x[1]^4*x[4] + 8*x[1]^3*x[2]*x[4] + 14*x[1]^2*x[2]^2*x[4] +
>        3*x[1]*x[2]^3*x[4] + 11*x[2]^4*x[4] + 7*x[1]^3*x[3]*x[4] +
>        3*x[1]^2*x[2]*x[3]*x[4] + 12*x[2]^3*x[3]*x[4] + 3*x[1]^2*x[3]^2*x[4] +
>        13*x[1]*x[2]*x[3]^2*x[4] + 3*x[2]^2*x[3]^2*x[4] + 7*x[1]*x[3]^3*x[4] +
>        2*x[2]*x[3]^3*x[4] + 7*x[3]^4*x[4] + 13*x[1]^3*x[4]^2 +
>        6*x[1]^2*x[2]*x[4]^2 + 6*x[1]*x[2]^2*x[4]^2 + 6*x[2]^3*x[4]^2 +
>        2*x[1]^2*x[3]*x[4]^2 + 15*x[1]*x[2]*x[3]*x[4]^2 + 14*x[2]^2*x[3]*x[4]^2
>        + 3*x[1]*x[3]^2*x[4]^2 + 16*x[2]*x[3]^2*x[4]^2 + 3*x[3]^3*x[4]^2 +
>         6*x[1]^2*x[4]^3 + 10*x[2]^2*x[4]^3 + 7*x[2]*x[3]*x[4]^3 + 13*x[1]*x[4]^4
>         + 5*x[2]*x[4]^4 + 15*x[3]*x[4]^4 + 13*x[4]^5 + 2*x[1]^4*x[5] +
>         6*x[1]^3*x[2]*x[5] + 12*x[1]^2*x[2]^2*x[5] + 12*x[1]*x[2]^3*x[5] +
>         2*x[2]^4*x[5] + 5*x[1]^3*x[3]*x[5] + 12*x[1]^2*x[2]*x[3]*x[5] +
>         7*x[1]*x[2]^2*x[3]*x[5] + 11*x[2]^3*x[3]*x[5] + 2*x[1]^2*x[3]^2*x[5] +
>         3*x[1]*x[2]*x[3]^2*x[5] + 7*x[2]^2*x[3]^2*x[5] + 16*x[1]*x[3]^3*x[5] +
>         3*x[2]*x[3]^3*x[5] + 13*x[3]^4*x[5] + 2*x[1]^2*x[2]*x[4]*x[5] +
>         12*x[1]*x[2]^2*x[4]*x[5] + 2*x[2]^3*x[4]*x[5] + 10*x[1]^2*x[3]*x[4]*x[5]
>         + 9*x[1]*x[2]*x[3]*x[4]*x[5] + 6*x[2]^2*x[3]*x[4]*x[5] +
>         x[1]*x[3]^2*x[4]*x[5] + 6*x[2]*x[3]^2*x[4]*x[5] + 15*x[3]^3*x[4]*x[5] +
>         2*x[1]^2*x[4]^2*x[5] + 14*x[1]*x[2]*x[4]^2*x[5] +
>         13*x[1]*x[3]*x[4]^2*x[5] + 13*x[2]*x[3]*x[4]^2*x[5] +
>         2*x[3]^2*x[4]^2*x[5] + 12*x[1]*x[4]^3*x[5] + 8*x[2]*x[4]^3*x[5] +
>         8*x[3]*x[4]^3*x[5] + x[4]^4*x[5] + 3*x[1]^3*x[5]^2 +
>         7*x[1]^2*x[2]*x[5]^2 + 4*x[1]^2*x[3]*x[5]^2 + 3*x[1]*x[2]*x[3]*x[5]^2 +
>         9*x[2]^2*x[3]*x[5]^2 + 14*x[1]*x[3]^2*x[5]^2 + 13*x[2]*x[3]^2*x[5]^2 +
>         15*x[3]^3*x[5]^2 + x[1]^2*x[4]*x[5]^2 + 14*x[1]*x[2]*x[4]*x[5]^2 +
>         5*x[2]^2*x[4]*x[5]^2 + 10*x[1]*x[3]*x[4]*x[5]^2 +
>         5*x[2]*x[3]*x[4]*x[5]^2 + 7*x[3]^2*x[4]*x[5]^2 + 13*x[1]*x[4]^2*x[5]^2 +
>         2*x[2]*x[4]^2*x[5]^2 + 9*x[3]*x[4]^2*x[5]^2 + 3*x[4]^3*x[5]^2 +
>         14*x[1]*x[2]*x[5]^3 + 12*x[2]^2*x[5]^3 + 6*x[1]*x[3]*x[5]^3 +
>         16*x[2]*x[3]*x[5]^3 + 8*x[3]^2*x[5]^3 + 3*x[1]*x[4]*x[5]^3 +
>         4*x[2]*x[4]*x[5]^3 + 11*x[3]*x[4]*x[5]^3 + 15*x[4]^2*x[5]^3 +
>         14*x[1]*x[5]^4 + 13*x[2]*x[5]^4 + 4*x[3]*x[5]^4 + 4*x[4]*x[5]^4 +
>         13*x[5]^5,
>     15*x[1]^3*x[2]*x[3] + 11*x[1]^2*x[2]^2*x[3] + 14*x[1]*x[2]^3*x[3] +
>         x[2]^4*x[3] + 2*x[1]^3*x[3]^2 + 11*x[1]*x[2]^2*x[3]^2 + 7*x[2]^3*x[3]^2
>         + 3*x[1]^2*x[3]^3 + 4*x[1]*x[2]*x[3]^3 + 13*x[2]^2*x[3]^3 + x[1]*x[3]^4
>         + 4*x[3]^5 + 2*x[1]^4*x[4] + 11*x[1]^3*x[2]*x[4] + 13*x[1]^2*x[2]^2*x[4]
>         + 4*x[1]*x[2]^3*x[4] + 16*x[2]^4*x[4] + 5*x[1]^3*x[3]*x[4] +
>         4*x[1]^2*x[2]*x[3]*x[4] + 10*x[1]*x[2]^2*x[3]*x[4] + 8*x[2]^3*x[3]*x[4]
>         + 5*x[1]^2*x[3]^2*x[4] + 14*x[1]*x[2]*x[3]^2*x[4] + 2*x[2]^2*x[3]^2*x[4]
>         + 15*x[1]*x[3]^3*x[4] + 13*x[3]^4*x[4] + 9*x[1]^3*x[4]^2 +
>         3*x[1]^2*x[2]*x[4]^2 + 10*x[1]*x[2]^2*x[4]^2 + 12*x[2]^3*x[4]^2 +
>         8*x[1]^2*x[3]*x[4]^2 + 14*x[1]*x[2]*x[3]*x[4]^2 + 3*x[2]^2*x[3]*x[4]^2 +
>         2*x[1]*x[3]^2*x[4]^2 + 5*x[2]*x[3]^2*x[4]^2 + 10*x[3]^3*x[4]^2 +
>         5*x[1]^2*x[4]^3 + x[1]*x[2]*x[4]^3 + 8*x[2]^2*x[4]^3 +
>         7*x[1]*x[3]*x[4]^3 + 10*x[2]*x[3]*x[4]^3 + 13*x[3]^2*x[4]^3 +
>         10*x[1]*x[4]^4 + 7*x[2]*x[4]^4 + 16*x[3]*x[4]^4 + 16*x[4]^5 +
>         8*x[1]^3*x[3]*x[5] + 5*x[1]^2*x[2]*x[3]*x[5] + x[1]*x[2]^2*x[3]*x[5] +
>         16*x[2]^3*x[3]*x[5] + 10*x[1]^2*x[3]^2*x[5] + 12*x[1]*x[2]*x[3]^2*x[5] +
>         9*x[2]^2*x[3]^2*x[5] + 15*x[1]*x[3]^3*x[5] + 13*x[2]*x[3]^3*x[5] +
>         4*x[3]^4*x[5] + 14*x[1]^3*x[4]*x[5] + x[1]^2*x[2]*x[4]*x[5] +
>         10*x[1]*x[2]^2*x[4]*x[5] + 11*x[2]^3*x[4]*x[5] + 5*x[1]^2*x[3]*x[4]*x[5]
>         + 12*x[1]*x[2]*x[3]*x[4]*x[5] + 7*x[2]^2*x[3]*x[4]*x[5] +
>         5*x[1]*x[3]^2*x[4]*x[5] + 3*x[3]^3*x[4]*x[5] + 2*x[1]^2*x[4]^2*x[5] +
>         5*x[2]^2*x[4]^2*x[5] + 2*x[2]*x[3]*x[4]^2*x[5] + 8*x[3]^2*x[4]^2*x[5] +
>         x[1]*x[4]^3*x[5] + 5*x[2]*x[4]^3*x[5] + 3*x[3]*x[4]^3*x[5] +
>         14*x[4]^4*x[5] + 16*x[1]^2*x[3]*x[5]^2 + 4*x[1]*x[2]*x[3]*x[5]^2 +
>         11*x[2]^2*x[3]*x[5]^2 + 9*x[1]*x[3]^2*x[5]^2 + 16*x[2]*x[3]^2*x[5]^2 +
>         8*x[3]^3*x[5]^2 + 8*x[1]^2*x[4]*x[5]^2 + 11*x[1]*x[2]*x[4]*x[5]^2 +
>         3*x[2]^2*x[4]*x[5]^2 + 6*x[1]*x[3]*x[4]*x[5]^2 + 9*x[2]*x[3]*x[4]*x[5]^2
>         + 5*x[3]^2*x[4]*x[5]^2 + 15*x[1]*x[4]^2*x[5]^2 + 2*x[2]*x[4]^2*x[5]^2 +
>         8*x[3]*x[4]^2*x[5]^2 + 14*x[4]^3*x[5]^2 + x[1]*x[3]*x[5]^3 +
>         15*x[2]*x[3]*x[5]^3 + 10*x[3]^2*x[5]^3 + 11*x[1]*x[4]*x[5]^3 +
>         8*x[2]*x[4]*x[5]^3 + 15*x[3]*x[4]*x[5]^3 + 15*x[4]^2*x[5]^3 +
>         6*x[3]*x[5]^4 + 3*x[4]*x[5]^4,
>     9*x[1]^4*x[3] + 14*x[1]^3*x[2]*x[3] + 5*x[1]^2*x[2]^2*x[3] +
>         2*x[1]*x[2]^3*x[3] + 2*x[1]^3*x[3]^2 + 7*x[1]^2*x[2]*x[3]^2 +
>         5*x[1]*x[2]^2*x[3]^2 + 7*x[2]^3*x[3]^2 + 9*x[1]^2*x[3]^3 +
>         12*x[1]*x[2]*x[3]^3 + 2*x[2]^2*x[3]^3 + 9*x[1]*x[3]^4 + 2*x[2]*x[3]^4 +
>         x[3]^5 + 3*x[1]^3*x[3]*x[4] + 5*x[1]^2*x[2]*x[3]*x[4] +
>         7*x[1]*x[2]^2*x[3]*x[4] + 13*x[2]^3*x[3]*x[4] + 11*x[1]^2*x[3]^2*x[4] +
>         4*x[1]*x[2]*x[3]^2*x[4] + 11*x[2]^2*x[3]^2*x[4] + 14*x[1]*x[3]^3*x[4] +
>         16*x[2]*x[3]^3*x[4] + 15*x[1]^2*x[3]*x[4]^2 + 11*x[1]*x[2]*x[3]*x[4]^2 +
>         5*x[2]^2*x[3]*x[4]^2 + 6*x[1]*x[3]^2*x[4]^2 + 9*x[2]*x[3]^2*x[4]^2 +
>         16*x[3]^3*x[4]^2 + 9*x[2]*x[3]*x[4]^3 + 15*x[3]^2*x[4]^3 +
>         14*x[3]*x[4]^4 + 2*x[1]^4*x[5] + 11*x[1]^3*x[2]*x[5] +
>         13*x[1]^2*x[2]^2*x[5] + 4*x[1]*x[2]^3*x[5] + 16*x[2]^4*x[5] +
>         2*x[1]^3*x[3]*x[5] + 13*x[1]^2*x[2]*x[3]*x[5] + 9*x[1]*x[2]^2*x[3]*x[5]
>         + 5*x[2]^3*x[3]*x[5] + 5*x[1]^2*x[3]^2*x[5] + 3*x[1]*x[2]*x[3]^2*x[5] +
>         8*x[2]^2*x[3]^2*x[5] + x[1]*x[3]^3*x[5] + 7*x[2]*x[3]^3*x[5] +
>         3*x[3]^4*x[5] + 9*x[1]^3*x[4]*x[5] + 3*x[1]^2*x[2]*x[4]*x[5] +
>         10*x[1]*x[2]^2*x[4]*x[5] + 12*x[2]^3*x[4]*x[5] +
>         10*x[1]^2*x[3]*x[4]*x[5] + 12*x[1]*x[2]*x[3]*x[4]*x[5] +
>         3*x[2]^2*x[3]*x[4]*x[5] + 11*x[1]*x[3]^2*x[4]*x[5] +
>         5*x[2]*x[3]^2*x[4]*x[5] + 7*x[3]^3*x[4]*x[5] + 5*x[1]^2*x[4]^2*x[5] +
>         x[1]*x[2]*x[4]^2*x[5] + 8*x[2]^2*x[4]^2*x[5] + 4*x[1]*x[3]*x[4]^2*x[5] +
>         16*x[2]*x[3]*x[4]^2*x[5] + 8*x[3]^2*x[4]^2*x[5] + 10*x[1]*x[4]^3*x[5] +
>         7*x[2]*x[4]^3*x[5] + 2*x[3]*x[4]^3*x[5] + 16*x[4]^4*x[5] +
>         14*x[1]^3*x[5]^2 + x[1]^2*x[2]*x[5]^2 + 10*x[1]*x[2]^2*x[5]^2 +
>         11*x[2]^3*x[5]^2 + 12*x[1]^2*x[3]*x[5]^2 + 12*x[1]*x[2]*x[3]*x[5]^2 +
>         13*x[2]^2*x[3]*x[5]^2 + 4*x[1]*x[3]^2*x[5]^2 + 15*x[2]*x[3]^2*x[5]^2 +
>         10*x[3]^3*x[5]^2 + 2*x[1]^2*x[4]*x[5]^2 + 5*x[2]^2*x[4]*x[5]^2 +
>         6*x[1]*x[3]*x[4]*x[5]^2 + 5*x[2]*x[3]*x[4]*x[5]^2 + 7*x[3]^2*x[4]*x[5]^2
>         + x[1]*x[4]^2*x[5]^2 + 5*x[2]*x[4]^2*x[5]^2 + x[3]*x[4]^2*x[5]^2 +
>         14*x[4]^3*x[5]^2 + 8*x[1]^2*x[5]^3 + 11*x[1]*x[2]*x[5]^3 +
>         3*x[2]^2*x[5]^3 + 9*x[1]*x[3]*x[5]^3 + 6*x[2]*x[3]*x[5]^3 +
>         13*x[3]^2*x[5]^3 + 15*x[1]*x[4]*x[5]^3 + 2*x[2]*x[4]*x[5]^3 +
>         5*x[3]*x[4]*x[5]^3 + 14*x[4]^2*x[5]^3 + 11*x[1]*x[5]^4 + 8*x[2]*x[5]^4 +
>         5*x[3]*x[5]^4 + 15*x[4]*x[5]^4 + 3*x[5]^5,
>     13*x[1]^4*x[3] + 8*x[1]^3*x[2]*x[3] + 14*x[1]^2*x[2]^2*x[3] +
>         3*x[1]*x[2]^3*x[3] + 11*x[2]^4*x[3] + 7*x[1]^3*x[3]^2 +
>         3*x[1]^2*x[2]*x[3]^2 + 12*x[2]^3*x[3]^2 + 3*x[1]^2*x[3]^3 +
>         13*x[1]*x[2]*x[3]^3 + 3*x[2]^2*x[3]^3 + 7*x[1]*x[3]^4 + 2*x[2]*x[3]^4 +
>         7*x[3]^5 + 13*x[1]^3*x[3]*x[4] + 6*x[1]^2*x[2]*x[3]*x[4] +
>        6*x[1]*x[2]^2*x[3]*x[4] + 6*x[2]^3*x[3]*x[4] + 2*x[1]^2*x[3]^2*x[4] +
>        15*x[1]*x[2]*x[3]^2*x[4] + 14*x[2]^2*x[3]^2*x[4] + 3*x[1]*x[3]^3*x[4] +
>        16*x[2]*x[3]^3*x[4] + 3*x[3]^4*x[4] + 6*x[1]^2*x[3]*x[4]^2 +
>        10*x[2]^2*x[3]*x[4]^2 + 7*x[2]*x[3]^2*x[4]^2 + 13*x[1]*x[3]*x[4]^3 +
>        5*x[2]*x[3]*x[4]^3 + 15*x[3]^2*x[4]^3 + 13*x[3]*x[4]^4 + 15*x[1]^4*x[5]
>        + 15*x[1]^3*x[2]*x[5] + 2*x[1]^2*x[2]^2*x[5] + 16*x[1]*x[2]^3*x[5] +
>        16*x[2]^4*x[5] + 14*x[1]^3*x[3]*x[5] + 4*x[1]^2*x[2]*x[3]*x[5] +
>        10*x[1]*x[2]^2*x[3]*x[5] + 4*x[2]^3*x[3]*x[5] + 8*x[1]^2*x[3]^2*x[5] +
>        5*x[1]*x[2]*x[3]^2*x[5] + 11*x[2]^2*x[3]^2*x[5] + 12*x[2]*x[3]^3*x[5] +
>        2*x[3]^4*x[5] + 15*x[1]^2*x[2]*x[4]*x[5] + 6*x[2]^3*x[4]*x[5] +
>        9*x[1]^2*x[3]*x[4]*x[5] + 9*x[1]*x[2]*x[3]*x[4]*x[5] +
>        15*x[2]^2*x[3]*x[4]*x[5] + 14*x[1]*x[3]^2*x[4]*x[5] +
>        13*x[2]*x[3]^2*x[4]*x[5] + 6*x[3]^3*x[4]*x[5] + 4*x[1]^2*x[4]^2*x[5] +
>        7*x[1]*x[2]*x[4]^2*x[5] + 3*x[2]^2*x[4]^2*x[5] + 8*x[1]*x[3]*x[4]^2*x[5]
>        + 8*x[2]*x[3]*x[4]^2*x[5] + 3*x[3]^2*x[4]^2*x[5] + 15*x[1]*x[4]^3*x[5] +
>        3*x[2]*x[4]^3*x[5] + 8*x[3]*x[4]^3*x[5] + 2*x[4]^4*x[5] +
>        2*x[1]^3*x[5]^2 + 6*x[1]^2*x[2]*x[5]^2 + x[1]*x[2]^2*x[5]^2 +
>        7*x[2]^3*x[5]^2 + 3*x[1]^2*x[3]*x[5]^2 + 16*x[1]*x[2]*x[3]*x[5]^2 +
>        10*x[2]^2*x[3]*x[5]^2 + 10*x[1]*x[3]^2*x[5]^2 + 13*x[2]*x[3]^2*x[5]^2 +
>        2*x[3]^3*x[5]^2 + 4*x[1]^2*x[4]*x[5]^2 + x[1]*x[2]*x[4]*x[5]^2 +
>        9*x[2]^2*x[4]*x[5]^2 + 16*x[1]*x[3]*x[4]*x[5]^2 +
>        8*x[2]*x[3]*x[4]*x[5]^2 + 11*x[1]*x[4]^2*x[5]^2 + 11*x[2]*x[4]^2*x[5]^2
>        + 4*x[3]*x[4]^2*x[5]^2 + 10*x[4]^3*x[5]^2 + 10*x[1]^2*x[5]^3 +
>        14*x[2]^2*x[5]^3 + 16*x[1]*x[3]*x[5]^3 + 13*x[2]*x[3]*x[5]^3 +
>        15*x[3]^2*x[5]^3 + 16*x[1]*x[4]*x[5]^3 + 3*x[2]*x[4]*x[5]^3 +
>        4*x[3]*x[4]*x[5]^3 + 2*x[4]^2*x[5]^3 + x[1]*x[5]^4 + 7*x[2]*x[5]^4 +
>        7*x[4]*x[5]^4 + 2*x[5]^5,
>    15*x[1]^4*x[3] + 11*x[1]^3*x[2]*x[3] + 5*x[1]^2*x[2]^2*x[3] +
>        5*x[1]*x[2]^3*x[3] + 15*x[2]^4*x[3] + 12*x[1]^3*x[3]^2 +
>        5*x[1]^2*x[2]*x[3]^2 + 10*x[1]*x[2]^2*x[3]^2 + 6*x[2]^3*x[3]^2 +
>        15*x[1]^2*x[3]^3 + 14*x[1]*x[2]*x[3]^3 + 10*x[2]^2*x[3]^3 + x[1]*x[3]^4
>        + 14*x[2]*x[3]^4 + 4*x[3]^5 + 15*x[1]^4*x[4] + 15*x[1]^3*x[2]*x[4] +
>        2*x[1]^2*x[2]^2*x[4] + 16*x[1]*x[2]^3*x[4] + 16*x[2]^4*x[4] +
>        14*x[1]^3*x[3]*x[4] + 2*x[1]^2*x[2]*x[3]*x[4] + 15*x[1]*x[2]^2*x[3]*x[4]
>        + 2*x[2]^3*x[3]*x[4] + 15*x[1]^2*x[3]^2*x[4] + 13*x[1]*x[2]*x[3]^2*x[4]
>        + 5*x[2]^2*x[3]^2*x[4] + 16*x[1]*x[3]^3*x[4] + 6*x[2]*x[3]^3*x[4] +
>        4*x[3]^4*x[4] + 15*x[1]^2*x[2]*x[4]^2 + 6*x[2]^3*x[4]^2 +
>        7*x[1]^2*x[3]*x[4]^2 + 12*x[1]*x[2]*x[3]*x[4]^2 + 15*x[2]^2*x[3]*x[4]^2
>        + x[1]*x[3]^2*x[4]^2 + 4*x[3]^3*x[4]^2 + 4*x[1]^2*x[4]^3 +
>        7*x[1]*x[2]*x[4]^3 + 3*x[2]^2*x[4]^3 + 13*x[1]*x[3]*x[4]^3 +
>        12*x[3]^2*x[4]^3 + 15*x[1]*x[4]^4 + 3*x[2]*x[4]^4 + 7*x[3]*x[4]^4 +
>        2*x[4]^5 + 14*x[1]^3*x[3]*x[5] + 10*x[1]^2*x[2]*x[3]*x[5] +
>        13*x[1]^2*x[3]^2*x[5] + 14*x[1]*x[2]*x[3]^2*x[5] + 8*x[2]^2*x[3]^2*x[5]
>        + 3*x[1]*x[3]^3*x[5] + 4*x[2]*x[3]^3*x[5] + 2*x[3]^4*x[5] +
>        2*x[1]^3*x[4]*x[5] + 6*x[1]^2*x[2]*x[4]*x[5] + x[1]*x[2]^2*x[4]*x[5] +
>        7*x[2]^3*x[4]*x[5] + 2*x[1]^2*x[3]*x[4]*x[5] +
>        2*x[1]*x[2]*x[3]*x[4]*x[5] + 5*x[2]^2*x[3]*x[4]*x[5] +
>        8*x[2]*x[3]^2*x[4]*x[5] + 12*x[3]^3*x[4]*x[5] + 4*x[1]^2*x[4]^2*x[5] +
>        x[1]*x[2]*x[4]^2*x[5] + 9*x[2]^2*x[4]^2*x[5] + 3*x[1]*x[3]*x[4]^2*x[5] +
>        6*x[2]*x[3]*x[4]^2*x[5] + 8*x[3]^2*x[4]^2*x[5] + 11*x[1]*x[4]^3*x[5] +
>        11*x[2]*x[4]^3*x[5] + x[3]*x[4]^3*x[5] + 10*x[4]^4*x[5] +
>        3*x[1]*x[2]*x[3]*x[5]^2 + 5*x[2]^2*x[3]*x[5]^2 + 11*x[1]*x[3]^2*x[5]^2 +
>        x[2]*x[3]^2*x[5]^2 + 9*x[3]^3*x[5]^2 + 10*x[1]^2*x[4]*x[5]^2 +
>        14*x[2]^2*x[4]*x[5]^2 + 13*x[1]*x[3]*x[4]*x[5]^2 +
>        9*x[2]*x[3]*x[4]*x[5]^2 + 4*x[3]^2*x[4]*x[5]^2 + 16*x[1]*x[4]^2*x[5]^2 +
>        3*x[2]*x[4]^2*x[5]^2 + 6*x[3]*x[4]^2*x[5]^2 + 2*x[4]^3*x[5]^2 +
>        3*x[1]*x[3]*x[5]^3 + 4*x[2]*x[3]*x[5]^3 + 13*x[3]^2*x[5]^3 +
>        x[1]*x[4]*x[5]^3 + 7*x[2]*x[4]*x[5]^3 + 13*x[3]*x[4]*x[5]^3 +
>        7*x[4]^2*x[5]^3 + 4*x[3]*x[5]^4 + 2*x[4]*x[5]^4,
>    15*x[1]^3*x[2]^2 + 11*x[1]^2*x[2]^3 + 14*x[1]*x[2]^4 + x[2]^5 +
>        2*x[1]^3*x[2]*x[3] + 11*x[1]*x[2]^3*x[3] + 7*x[2]^4*x[3] +
>        3*x[1]^2*x[2]*x[3]^2 + 4*x[1]*x[2]^2*x[3]^2 + 13*x[2]^3*x[3]^2 +
>        x[1]*x[2]*x[3]^3 + 4*x[2]*x[3]^4 + 16*x[1]^4*x[4] + 2*x[1]^3*x[2]*x[4] +
>        7*x[1]^2*x[2]^2*x[4] + 4*x[1]*x[2]^3*x[4] + 4*x[2]^4*x[4] +
>        9*x[1]^3*x[3]*x[4] + x[1]^2*x[2]*x[3]*x[4] + 9*x[1]*x[2]^2*x[3]*x[4] +
>        4*x[2]^3*x[3]*x[4] + 7*x[1]^2*x[3]^2*x[4] + 5*x[1]*x[2]*x[3]^2*x[4] +
>        11*x[2]^2*x[3]^2*x[4] + 15*x[1]*x[3]^3*x[4] + 15*x[2]*x[3]^3*x[4] +
>        x[3]^4*x[4] + 9*x[1]^2*x[2]*x[4]^2 + 16*x[1]*x[2]^2*x[4]^2 +
>        9*x[2]^3*x[4]^2 + 3*x[1]^2*x[3]*x[4]^2 + 2*x[1]*x[2]*x[3]*x[4]^2 +
>        14*x[2]^2*x[3]*x[4]^2 + 11*x[1]*x[3]^2*x[4]^2 + 16*x[2]*x[3]^2*x[4]^2 +
>        4*x[3]^3*x[4]^2 + x[1]^2*x[4]^3 + 8*x[1]*x[2]*x[4]^3 + 14*x[2]^2*x[4]^3
>        + 3*x[1]*x[3]*x[4]^3 + 16*x[2]*x[3]*x[4]^3 + 12*x[3]^2*x[4]^3 +
>        7*x[1]*x[4]^4 + 5*x[2]*x[4]^4 + 4*x[3]*x[4]^4 + 2*x[4]^5 +
>        8*x[1]^3*x[2]*x[5] + 5*x[1]^2*x[2]^2*x[5] + x[1]*x[2]^3*x[5] +
>        16*x[2]^4*x[5] + 10*x[1]^2*x[2]*x[3]*x[5] + 12*x[1]*x[2]^2*x[3]*x[5] +
>        9*x[2]^3*x[3]*x[5] + 15*x[1]*x[2]*x[3]^2*x[5] + 13*x[2]^2*x[3]^2*x[5] +
>        4*x[2]*x[3]^3*x[5] + 8*x[1]^3*x[4]*x[5] + 16*x[1]^2*x[2]*x[4]*x[5] +
>        11*x[1]*x[2]^2*x[4]*x[5] + 8*x[2]^3*x[4]*x[5] + 10*x[1]^2*x[3]*x[4]*x[5]
>        + 15*x[1]*x[2]*x[3]*x[4]*x[5] + 4*x[2]^2*x[3]*x[4]*x[5] +
>        9*x[1]*x[3]^2*x[4]*x[5] + 16*x[2]*x[3]^2*x[4]*x[5] + 11*x[3]^3*x[4]*x[5]
>        + 4*x[1]^2*x[4]^2*x[5] + 6*x[1]*x[2]*x[4]^2*x[5] + 10*x[2]^2*x[4]^2*x[5]
>        + 11*x[1]*x[3]*x[4]^2*x[5] + 11*x[2]*x[3]*x[4]^2*x[5] +
>        14*x[3]^2*x[4]^2*x[5] + 10*x[1]*x[4]^3*x[5] + 6*x[2]*x[4]^3*x[5] +
>        5*x[3]*x[4]^3*x[5] + 4*x[4]^4*x[5] + 16*x[1]^2*x[2]*x[5]^2 +
>        4*x[1]*x[2]^2*x[5]^2 + 11*x[2]^3*x[5]^2 + 9*x[1]*x[2]*x[3]*x[5]^2 +
>        16*x[2]^2*x[3]*x[5]^2 + 8*x[2]*x[3]^2*x[5]^2 + 3*x[1]^2*x[4]*x[5]^2 +
>        10*x[1]*x[2]*x[4]*x[5]^2 + 9*x[2]^2*x[4]*x[5]^2 +
>        10*x[1]*x[3]*x[4]*x[5]^2 + 11*x[2]*x[3]*x[4]*x[5]^2 +
>        11*x[3]^2*x[4]*x[5]^2 + 3*x[1]*x[4]^2*x[5]^2 + 14*x[2]*x[4]^2*x[5]^2 +
>        7*x[3]*x[4]^2*x[5]^2 + 3*x[4]^3*x[5]^2 + x[1]*x[2]*x[5]^3 +
>        15*x[2]^2*x[5]^3 + 10*x[2]*x[3]*x[5]^3 + x[1]*x[4]*x[5]^3 +
>        2*x[2]*x[4]*x[5]^3 + 5*x[3]*x[4]*x[5]^3 + 6*x[4]^2*x[5]^3 +
>        6*x[2]*x[5]^4 + 16*x[4]*x[5]^4,
>    2*x[1]^4*x[2] + 2*x[1]^3*x[2]^2 + 15*x[1]^2*x[2]^3 + x[1]*x[2]^4 + x[2]^5 +
>        16*x[1]^4*x[3] + 8*x[1]^3*x[2]*x[3] + 7*x[1]^2*x[2]^2*x[3] +
>        10*x[1]*x[2]^3*x[3] + 10*x[2]^4*x[3] + 10*x[1]^3*x[3]^2 +
>         11*x[1]^2*x[2]*x[3]^2 + 9*x[1]*x[2]^2*x[3]^2 + 10*x[2]^3*x[3]^2 +
>         7*x[1]^2*x[3]^3 + 10*x[1]*x[2]*x[3]^3 + 4*x[2]^2*x[3]^3 + 13*x[1]*x[3]^4
>         + 3*x[3]^5 + 2*x[1]^2*x[2]^2*x[4] + 11*x[2]^4*x[4] + 5*x[1]^3*x[3]*x[4]
>         + 15*x[1]^2*x[2]*x[3]*x[4] + 16*x[1]*x[2]^2*x[3]*x[4] +
>         4*x[2]^3*x[3]*x[4] + 16*x[1]^2*x[3]^2*x[4] + 12*x[1]*x[2]*x[3]^2*x[4] +
>         4*x[2]^2*x[3]^2*x[4] + 15*x[1]*x[3]^3*x[4] + 14*x[2]*x[3]^3*x[4] +
>         5*x[3]^4*x[4] + 13*x[1]^2*x[2]*x[4]^2 + 10*x[1]*x[2]^2*x[4]^2 +
>         14*x[2]^3*x[4]^2 + 10*x[1]^2*x[3]*x[4]^2 + 9*x[1]*x[2]*x[3]*x[4]^2 +
>         2*x[2]^2*x[3]*x[4]^2 + x[1]*x[3]^2*x[4]^2 + 15*x[2]*x[3]^2*x[4]^2 +
>         15*x[3]^3*x[4]^2 + 2*x[1]*x[2]*x[4]^3 + 14*x[2]^2*x[4]^3 +
>         15*x[1]*x[3]*x[4]^3 + 6*x[2]*x[3]*x[4]^3 + 3*x[3]^2*x[4]^3 +
>         15*x[2]*x[4]^4 + 13*x[3]*x[4]^4 + 15*x[1]^3*x[2]*x[5] +
>         11*x[1]^2*x[2]^2*x[5] + 16*x[1]*x[2]^3*x[5] + 10*x[2]^4*x[5] +
>         4*x[1]^3*x[3]*x[5] + 10*x[1]^2*x[2]*x[3]*x[5] + 4*x[2]^3*x[3]*x[5] +
>         3*x[1]^2*x[3]^2*x[5] + 6*x[1]*x[2]*x[3]^2*x[5] + 15*x[2]^2*x[3]^2*x[5] +
>         8*x[1]*x[3]^3*x[5] + 8*x[2]*x[3]^3*x[5] + x[3]^4*x[5] +
>         13*x[1]^2*x[2]*x[4]*x[5] + 16*x[1]*x[2]^2*x[4]*x[5] + 8*x[2]^3*x[4]*x[5]
>         + 5*x[1]^2*x[3]*x[4]*x[5] + 16*x[1]*x[2]*x[3]*x[4]*x[5] +
>         x[2]^2*x[3]*x[4]*x[5] + 10*x[1]*x[3]^2*x[4]*x[5] +
>         9*x[2]*x[3]^2*x[4]*x[5] + 12*x[3]^3*x[4]*x[5] + 6*x[1]*x[2]*x[4]^2*x[5]
>         + 6*x[2]^2*x[4]^2*x[5] + 12*x[1]*x[3]*x[4]^2*x[5] +
>         16*x[2]*x[3]*x[4]^2*x[5] + 6*x[3]^2*x[4]^2*x[5] + 7*x[2]*x[4]^3*x[5] +
>         15*x[3]*x[4]^3*x[5] + 7*x[1]^2*x[2]*x[5]^2 + 3*x[2]^3*x[5]^2 +
>         3*x[1]^2*x[3]*x[5]^2 + 11*x[1]*x[2]*x[3]*x[5]^2 + x[2]^2*x[3]*x[5]^2 +
>         12*x[1]*x[3]^2*x[5]^2 + 8*x[2]*x[3]^2*x[5]^2 + 4*x[3]^3*x[5]^2 +
>         x[1]*x[2]*x[4]*x[5]^2 + 14*x[2]^2*x[4]*x[5]^2 + 8*x[1]*x[3]*x[4]*x[5]^2
>         + 13*x[2]*x[3]*x[4]*x[5]^2 + 4*x[3]^2*x[4]*x[5]^2 +
>         15*x[2]*x[4]^2*x[5]^2 + 16*x[1]*x[2]*x[5]^3 + 10*x[2]^2*x[5]^3 +
>         9*x[1]*x[3]*x[5]^3 + 14*x[2]*x[3]*x[5]^3 + 12*x[3]^2*x[5]^3 +
>         10*x[2]*x[4]*x[5]^3 + x[3]*x[4]*x[5]^3 + 15*x[2]*x[5]^4 + 9*x[3]*x[5]^4,
>     9*x[1]^4*x[2] + 14*x[1]^3*x[2]^2 + 5*x[1]^2*x[2]^3 + 2*x[1]*x[2]^4 +
>         2*x[1]^3*x[2]*x[3] + 7*x[1]^2*x[2]^2*x[3] + 5*x[1]*x[2]^3*x[3] +
>         7*x[2]^4*x[3] + 9*x[1]^2*x[2]*x[3]^2 + 12*x[1]*x[2]^2*x[3]^2 +
>         2*x[2]^3*x[3]^2 + 9*x[1]*x[2]*x[3]^3 + 2*x[2]^2*x[3]^3 + x[2]*x[3]^4 +
>         3*x[1]^3*x[2]*x[4] + 5*x[1]^2*x[2]^2*x[4] + 7*x[1]*x[2]^3*x[4] +
>         13*x[2]^4*x[4] + 11*x[1]^2*x[2]*x[3]*x[4] + 4*x[1]*x[2]^2*x[3]*x[4] +
>         11*x[2]^3*x[3]*x[4] + 14*x[1]*x[2]*x[3]^2*x[4] + 16*x[2]^2*x[3]^2*x[4] +
>         15*x[1]^2*x[2]*x[4]^2 + 11*x[1]*x[2]^2*x[4]^2 + 5*x[2]^3*x[4]^2 +
>         6*x[1]*x[2]*x[3]*x[4]^2 + 9*x[2]^2*x[3]*x[4]^2 + 16*x[2]*x[3]^2*x[4]^2 +
>         9*x[2]^2*x[4]^3 + 15*x[2]*x[3]*x[4]^3 + 14*x[2]*x[4]^4 + 16*x[1]^4*x[5]
>         + 16*x[1]^3*x[2]*x[5] + 16*x[1]^2*x[2]^2*x[5] + 3*x[1]*x[2]^3*x[5] +
>         x[2]^4*x[5] + 9*x[1]^3*x[3]*x[5] + x[1]^2*x[2]*x[3]*x[5] +
>         15*x[1]*x[2]^2*x[3]*x[5] + 10*x[2]^3*x[3]*x[5] + 7*x[1]^2*x[3]^2*x[5] +
>         8*x[1]*x[2]*x[3]^2*x[5] + x[2]^2*x[3]^2*x[5] + 15*x[1]*x[3]^3*x[5] +
>         5*x[2]*x[3]^3*x[5] + x[3]^4*x[5] + 11*x[1]^2*x[2]*x[4]*x[5] +
>         14*x[1]*x[2]^2*x[4]*x[5] + 9*x[2]^3*x[4]*x[5] + 3*x[1]^2*x[3]*x[4]*x[5]
>         + 11*x[1]*x[2]*x[3]*x[4]*x[5] + 14*x[2]^2*x[3]*x[4]*x[5] +
>         11*x[1]*x[3]^2*x[4]*x[5] + 13*x[2]*x[3]^2*x[4]*x[5] + 4*x[3]^3*x[4]*x[5]
>         + x[1]^2*x[4]^2*x[5] + 5*x[1]*x[2]*x[4]^2*x[5] + 3*x[2]^2*x[4]^2*x[5] +
>         3*x[1]*x[3]*x[4]^2*x[5] + 11*x[2]*x[3]*x[4]^2*x[5] +
>         12*x[3]^2*x[4]^2*x[5] + 7*x[1]*x[4]^3*x[5] + 8*x[2]*x[4]^3*x[5] +
>         4*x[3]*x[4]^3*x[5] + 2*x[4]^4*x[5] + 8*x[1]^3*x[5]^2 +
>         6*x[1]^2*x[2]*x[5]^2 + 11*x[1]*x[2]^2*x[5]^2 + 14*x[2]^3*x[5]^2 +
>         10*x[1]^2*x[3]*x[5]^2 + 14*x[1]*x[2]*x[3]*x[5]^2 + 2*x[2]^2*x[3]*x[5]^2
>         + 9*x[1]*x[3]^2*x[5]^2 + 6*x[2]*x[3]^2*x[5]^2 + 11*x[3]^3*x[5]^2 +
>         4*x[1]^2*x[4]*x[5]^2 + 12*x[1]*x[2]*x[4]*x[5]^2 + 13*x[2]^2*x[4]*x[5]^2
>         + 11*x[1]*x[3]*x[4]*x[5]^2 + 10*x[2]*x[3]*x[4]*x[5]^2 +
>         14*x[3]^2*x[4]*x[5]^2 + 10*x[1]*x[4]^2*x[5]^2 + 4*x[2]*x[4]^2*x[5]^2 +
>         5*x[3]*x[4]^2*x[5]^2 + 4*x[4]^3*x[5]^2 + 3*x[1]^2*x[5]^3 +
>         13*x[1]*x[2]*x[5]^3 + 6*x[2]^2*x[5]^3 + 10*x[1]*x[3]*x[5]^3 +
>         2*x[2]*x[3]*x[5]^3 + 11*x[3]^2*x[5]^3 + 3*x[1]*x[4]*x[5]^3 +
>         11*x[2]*x[4]*x[5]^3 + 7*x[3]*x[4]*x[5]^3 + 3*x[4]^2*x[5]^3 + x[1]*x[5]^4
>         + 9*x[2]*x[5]^4 + 5*x[3]*x[5]^4 + 6*x[4]*x[5]^4 + 16*x[5]^5,
>     13*x[1]^4*x[2] + 8*x[1]^3*x[2]^2 + 14*x[1]^2*x[2]^3 + 3*x[1]*x[2]^4 +
>         11*x[2]^5 + 7*x[1]^3*x[2]*x[3] + 3*x[1]^2*x[2]^2*x[3] + 12*x[2]^4*x[3] +
>         3*x[1]^2*x[2]*x[3]^2 + 13*x[1]*x[2]^2*x[3]^2 + 3*x[2]^3*x[3]^2 +
>         7*x[1]*x[2]*x[3]^3 + 2*x[2]^2*x[3]^3 + 7*x[2]*x[3]^4 +
>         13*x[1]^3*x[2]*x[4] + 6*x[1]^2*x[2]^2*x[4] + 6*x[1]*x[2]^3*x[4] +
>         6*x[2]^4*x[4] + 2*x[1]^2*x[2]*x[3]*x[4] + 15*x[1]*x[2]^2*x[3]*x[4] +
>         14*x[2]^3*x[3]*x[4] + 3*x[1]*x[2]*x[3]^2*x[4] + 16*x[2]^2*x[3]^2*x[4] +
>         3*x[2]*x[3]^3*x[4] + 6*x[1]^2*x[2]*x[4]^2 + 10*x[2]^3*x[4]^2 +
>         7*x[2]^2*x[3]*x[4]^2 + 13*x[1]*x[2]*x[4]^3 + 5*x[2]^2*x[4]^3 +
>         15*x[2]*x[3]*x[4]^3 + 13*x[2]*x[4]^4 + 16*x[1]^4*x[5] +
>         5*x[1]^3*x[2]*x[5] + 11*x[1]^2*x[2]^2*x[5] + 3*x[1]*x[2]^3*x[5] +
>         14*x[2]^4*x[5] + 10*x[1]^3*x[3]*x[5] + 2*x[1]^2*x[2]*x[3]*x[5] +
>         14*x[1]*x[2]^2*x[3]*x[5] + 4*x[2]^3*x[3]*x[5] + 7*x[1]^2*x[3]^2*x[5] +
>         10*x[1]*x[2]*x[3]^2*x[5] + 16*x[2]^2*x[3]^2*x[5] + 13*x[1]*x[3]^3*x[5] +
>         2*x[2]*x[3]^3*x[5] + 3*x[3]^4*x[5] + 5*x[1]^3*x[4]*x[5] +
>         7*x[1]^2*x[2]*x[4]*x[5] + 8*x[1]*x[2]^2*x[4]*x[5] + 2*x[2]^3*x[4]*x[5] +
>         16*x[1]^2*x[3]*x[4]*x[5] + 9*x[1]*x[2]*x[3]*x[4]*x[5] +
>         15*x[1]*x[3]^2*x[4]*x[5] + 3*x[2]*x[3]^2*x[4]*x[5] + 5*x[3]^3*x[4]*x[5]
>         + 10*x[1]^2*x[4]^2*x[5] + 10*x[2]^2*x[4]^2*x[5] + x[1]*x[3]*x[4]^2*x[5]
>         + x[2]*x[3]*x[4]^2*x[5] + 15*x[3]^2*x[4]^2*x[5] + 15*x[1]*x[4]^3*x[5] +
>         14*x[2]*x[4]^3*x[5] + 3*x[3]*x[4]^3*x[5] + 13*x[4]^4*x[5] +
>         4*x[1]^3*x[5]^2 + 13*x[1]^2*x[2]*x[5]^2 + 16*x[1]*x[2]^2*x[5]^2 +
>         14*x[2]^3*x[5]^2 + 3*x[1]^2*x[3]*x[5]^2 + 16*x[1]*x[2]*x[3]*x[5]^2 +
>         11*x[2]^2*x[3]*x[5]^2 + 8*x[1]*x[3]^2*x[5]^2 + 10*x[2]*x[3]^2*x[5]^2 +
>         x[3]^3*x[5]^2 + 5*x[1]^2*x[4]*x[5]^2 + 15*x[1]*x[2]*x[4]*x[5]^2 +
>         9*x[2]^2*x[4]*x[5]^2 + 10*x[1]*x[3]*x[4]*x[5]^2 +
>         9*x[2]*x[3]*x[4]*x[5]^2 + 12*x[3]^2*x[4]*x[5]^2 + 12*x[1]*x[4]^2*x[5]^2
>         + 3*x[2]*x[4]^2*x[5]^2 + 6*x[3]*x[4]^2*x[5]^2 + 15*x[4]^3*x[5]^2 +
>         3*x[1]^2*x[5]^3 + 10*x[1]*x[2]*x[5]^3 + 14*x[2]^2*x[5]^3 +
>         12*x[1]*x[3]*x[5]^3 + 6*x[2]*x[3]*x[5]^3 + 4*x[3]^2*x[5]^3 +
>         8*x[1]*x[4]*x[5]^3 + 4*x[3]*x[4]*x[5]^3 + 9*x[1]*x[5]^4 + 14*x[2]*x[5]^4
>       + 12*x[3]*x[5]^4 + x[4]*x[5]^4 + 9*x[5]^5]);
```

We check a few of the invariants of X.

```
> Dimension(X);
2
> IsNonsingular(X);
true
> ArithmeticGenus(X);
0
> // Get the sectional genus of X -- ie the genus of a hyperplane section.
> ArithmeticGenus(X meet Scheme(P,P.1));
9
```

Now we construct the canonical sheaf and hyperplane sheaf and check intersection numbers.

```
> KX := CanonicalSheaf(X);
> HX := StructureSheaf(X,1); // hyperplane sheaf
> IntersectionPairing(HX,HX); // should be 10 = Degree(X)
10
> Degree(X);
10
> IntersectionPairing(KX,HX); // should be 6
6
> IntersectionPairing(KX,KX); // should be -9 : lots of exceptional curves!
-9
```

We now get the adjunction map as a divisor map, compute its image X_1 and check some of the invariants of X_1 as well as its corresponding intersection numbers.

```
> mp1,X1 := DivisorMap(Twist(KX,1));
> Dimension(Ambient(X1)); Dimension(X1);
8
2
> KX1 := CanonicalSheaf(X1);
> HX1 := StructureSheaf(X1,1); // hyperplane sheaf of X1
> IntersectionPairing(HX1,HX1); // should be 13 = degree X1
13
> IntersectionPairing(KX1,HX1); // should be -3
-3
> IntersectionPairing(KX1,KX1); // should be -2 : fewer exceptional curves!
-2
```

We construct a second adjunction map to get X_2 and check it as above.

```
> mp2,X2 := DivisorMap(Twist(KX1,1));
> Dimension(Ambient(X2)); Dimension(X2);
5
2
> KX2 := CanonicalSheaf(X2);
> HX2 := StructureSheaf(X2,1); // hyperplane sheaf X2
> IntersectionPairing(HX2,HX2); // = degree X2 = 5
5
> IntersectionPairing(KX2,HX2); // should be -5
-5
> IntersectionPairing(KX2,KX2); // should be 5
5
```

Now X_2 should be a degree five Del Pezzo surface with $\mathcal{K}_X \simeq \mathcal{O}_X(-1)$. This last isomorphism can be verified by checking that there is a degree -2 isomorphism from $\mathcal{K}_X$ to $\mathcal{O}_X(1)$! The scheme X_2 is much simpler than X: it is defined by five degree 2 polynomials.

```
> boo,d := IsIsomorphicWithTwist(KX2,HX2);
> boo; d;
true
-2
> MinimalBasis(Ideal(X2));
Scheme over GF(17) defined by
y[1]^2 + y[3]^2 + y[1]*y[4] + 15*y[2]*y[4] + 8*y[3]*y[4] + 6*y[4]^2 +
    2*y[1]*y[5] + 12*y[2]*y[5] + y[3]*y[5] + 4*y[4]*y[5] + 4*y[5]^2 +
    6*y[1]*y[6] + 10*y[2]*y[6] + 7*y[3]*y[6] + 7*y[5]*y[6] + 16*y[6]^2,
y[1]*y[2] + 13*y[3]^2 + 3*y[1]*y[4] + 14*y[2]*y[4] + 13*y[3]*y[4] + 5*y[4]^2 +
    14*y[1]*y[5] + 10*y[2]*y[5] + 2*y[3]*y[5] + 9*y[4]*y[5] + 6*y[5]^2 +
    4*y[1]*y[6] + 13*y[2]*y[6] + 10*y[3]*y[6] + 3*y[4]*y[6] + y[5]*y[6] +
    12*y[6]^2,
y[2]^2 + 16*y[3]^2 + 15*y[1]*y[4] + 3*y[3]*y[4] + y[4]^2 + 10*y[1]*y[5] +
    12*y[2]*y[5] + 10*y[3]*y[5] + 11*y[4]*y[5] + 9*y[5]^2 + 5*y[1]*y[6] +
    3*y[2]*y[6] + 2*y[3]*y[6] + 15*y[4]*y[6] + 12*y[5]*y[6] + 5*y[6]^2,
y[1]*y[3] + 13*y[3]^2 + y[1]*y[4] + 11*y[3]*y[4] + y[4]^2 + 16*y[1]*y[5] +
    y[2]*y[5] + 15*y[3]*y[5] + 3*y[4]*y[5] + 7*y[1]*y[6] + 3*y[2]*y[6] +
    9*y[3]*y[6] + 10*y[4]*y[6] + 8*y[5]*y[6] + 6*y[6]^2,
y[2]*y[3] + 16*y[3]^2 + 14*y[1]*y[4] + 3*y[2]*y[4] + y[3]*y[4] + y[4]^2 +
    12*y[1]*y[5] + 9*y[3]*y[5] + 6*y[4]*y[5] + 2*y[5]^2 + 13*y[3]*y[6] +
    9*y[4]*y[6] + 13*y[5]*y[6] + 12*y[6]^2
```

Finally we get the composed map from X to X_2 and check that it is (birationally) invertible.

```
> mp1r := Restriction(mp1,X,X1);
> mp2r := Restriction(mp2,X1,X2);
> mpc := Expand(mp1r*mp2r);
> boo := IsInvertible(mpc);
> boo;
true
```

Example H113E7

In this example, we show how the sheaf machinery can be effectively used as an alternative method to normalise the projective coordinate ring of a normal, but not projectively normal, projective variety. Here the coordinate ring is locally normal at all primes except at the maximal homogeneous ideal.

Our chosen variety is C, an elliptic curve that has been embedded as a degree 8 subvariety of $\mathbf{P}^3$ over $\mathbf{Q}$. The curve C can be thought of as having been embedded in $\mathbf{P}^7$ by a complete linear system of degree 8 and then (isomorphically) projected down into $\mathbf{P}^3$. Such genus one curves embedded as degree 8 curves in $\mathbf{P}^3$ actually arise fairly naturally as models of homogeneous spaces arising in eight-descents.

We wish to recover the full embedding as a projective normal curve in $\mathbf{P}^7$. The coordinate ring of this is isomorphic to the normalisation of the coordinate ring of C in $\mathbf{P}^3$. From a sheaf-theoretic point of view, this is straightforward. The full embedding is the image of the divisor map corresponding to a hyperplane section of C or, equivalently, to the Serre twisting sheaf $\mathcal{O}_X(1)$. The maximal module of $\mathcal{O}_X(1)$ is isomorphic to the normalisation as an R-module, where R is the coordinate ring of C in $\mathbf{P}^3$, and it can be recovered as an algebra by taking the image of its associated divisor map. The global sections of $\mathcal{O}_X(1)$ correspond to the full Riemann-Roch space of the divisor on the abstract curve given by a certain hyperplane divisor on C.

This example also illustrates another interesting point. In situations similar to these, the dimension of the full space of global sections of the Serre twisting sheaf can be computed from cohomology of the coordinate ring R. However, it is faster in this case to explicitly compute the full maximal module of $\mathcal{O}_X(1)$, the zero-th graded part of this corresponding to the space of global sections and having the dimension of the zeroth cohomology group. In fact, though we only need to compute the dimension of this part, it is actually much quicker to compute the maximal module and compute its cohomology than to compute the cohomology of the original defining module, which is R twisted once. This probably reflects to some extent the fact that polynomial ring Groebner basis computations are much more highly tuned currently in MAGMA than the alternating algebra ones used in the cohomology computations. But the maximal module of a sheaf is generally a nicer object than a submodule with bits missing in the lower-graded pieces and has a smaller Castelnuevo-Mumford regularity etc. So, as we see in this example, it is often worth making sure that the maximal module of a sheaf is available before making cohomology calls.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> C := Curve(P,[ x^2*y^2 - 23/59*x*y^3 + 9/59*y^4 + 27/59*x^3*z - 23/59*x^2*y*
> z - 6/59*x*y^2*z + 6/59*y^3*z - 10/59*x^2*z^2 + 5/59*x*y*z^2 - 3/59*y^2*z^2 +
> 1/59*x*z^3 - 74/59*x^3*t + 115/59*x^2*y*t - 83/59*x*y^2*t + 3/59*y^3*t -
> 105/59*x^2*z*t + 1/59*x*y*z*t - 2/59*y^2*z*t + 36/59*x*z^2*t + 4/59*y*z^2*t -
> 3/59*z^3*t + 297/59*x^2*t^2 - 135/59*x*y*t^2 + 52/59*y^2*t^2 + 68/59*x*z*t^2 -
> 11/59*y*z*t^2 - 18/59*z^2*t^2 - 315/59*x*t^3 + 42/59*y*t^3 + 96/59*t^4,
> x^3*y - 833/354*x*y^3 - 11/236*y^4 - 1633/708*x^3*z - 4675/708*x^2*y*z -
> 2633/708*x*y^2*z - 27/236*y^3*z + 805/354*x^2*z^2 + 223/59*x*y*z^2 -
> 4/59*y^2*z^2 - 38/59*x*z^3 + 3359/708*x^3*t + 3811/354*x^2*y*t +
> 1445/708*x*y^2*t + 303/118*y^3*t - 715/177*x^2*z*t - 527/177*x*y*z*t +
> 211/118*y^2*z*t + 347/354*x*z^2*t - 195/236*y*z^2*t - 4/59*z^3*t -
> 127/236*x^2*t^2 - 8237/708*x*y*t^2 + 65/708*y^2*t^2 + 1973/708*x*z*t^2 +
> 123/59*y*z*t^2 - 24/59*z^2*t^2 - 1753/354*x*t^3 + 873/236*y*t^3 + 128/59*t^4,
> x^4 + 269/354*x*y^3 + 35/236*y^4 + 1849/708*x^3*z + 4255/708*x^2*y*z -
> 247/708*x*y^2*z + 43/236*y^3*z - 727/354*x^2*z^2 - 82/59*x*y*z^2 +
> 2/59*y^2*z^2 + 19/59*x*z^3 - 5603/708*x^3*t - 3469/354*x^2*y*t -
> 1637/708*x*y^2*t - 63/118*y^3*t + 328/177*x^2*z*t - 769/177*x*y*z*t -
> 17/118*y^2*z*t + 151/354*x*z^2*t + 127/236*y*z^2*t + 2/59*z^3*t +
> 1391/236*x^2*t^2 + 7865/708*x*y*t^2 + 823/708*y^2*t^2 - 1901/708*x*z*t^2 +
> 86/59*y*z*t^2 + 12/59*z^2*t^2 + 493/354*x*t^3 - 761/236*y*t^3 - 64/59*t^4]);
```

Next the hyperplane sheaf of C is constructed and the dimension of the space of global sections is confirmed to be 8 using `DimensionOfGlobalSections` (which also saturates the sheaf).

```
> OC1 := StructureSheaf(C,1);
> DimensionOfGlobalSections(OC1);
8
```

Finally, the projective normal embedding into P^7 is created and we check that the image X is defined by 20 quadrics.

```
> norm_mp, X := DivisorMap(OC1);
> ArithmeticGenus(X);
1
> B := MinimalBasis(Ideal(X));
> #B;
20
> [TotalDegree(f) : f in B];
[ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
```

113.10 Bibliography

[**DES93**] Decker, Ein, and Schreyer. Construction of surfaces in P4. *J. Algebraic Geometry*, 2:185–237, 1993.

[**Eis95**] David Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry*, volume 150 of *Graduate Texts in Mathematics.* Springer, New York–Berlin–Heidelberg, 1995.

[**Har77**] Robin Hartshorne. *Algebraic Geometry, GTM 52.* Springer, ASpringer, 1977.

[**HM73**] G. Horrocks and D. Mumford. A rank 2 vector bundle on $\mathbf{P}_4$ with 15,000 symmetries. *Topology*, 12:63–81, 1973.

[**Ser55**] J-P. Serre. Faisceaux Algebriques Coherents. *Ann. Maths.*, 61:197–278, 1955.

114 ALGEBRAIC CURVES

Chapter 114
ALGEBRAIC CURVES

114.1 First Examples

This chapter describes functions for constructing and studying algebraic curves. The first section below contains elementary examples to help with getting started. The biggest obstacle is being able to create geometric objects. After that one should be able to consult the later sections for off-the-shelf functions to apply to the curves.

Within Magma, curves are realised as a specialised type of scheme, themselves covered in Chapter 112. As schemes they may be defined over any ring, although most functions will require this to be at least a domain and often a field.

In previous versions of Magma, curves were restricted to lie in some plane for the dedicated functions below to apply to them. However, we have now generalised the curve definition to apply to any one-dimensional scheme. The general type is Crv and the plane curve now has sub-type CrvPln. As before, for the vast majority of the specialist functionality to apply to a curve, it has to be integral (reduced and irreducible) and defined over a field.

114.1.1 Ambients

One usually starts by defining an affine or projective space over some base ring in which our curves will live, although it is not absolutely necessary. Normal affine and projective spaces are the commonest, although product or weighted projective spaces may also be used. This space is often referred to as the *ambient space* and these are more fully described in the general chapter on schemes referred to above. A two-dimensional ambient is referred to as a plane. Plane curves - the ones of subtype CrvPln - are those whose ambient space is a plane (and these are defined by a single polynomial equation). If you intend later to create several curves and would like them to be taken to lie in the same space, then deliberately creating their common ambient space in advance is certainly the surest way. It is important to be aware that any two ambients that have been created independently will always be distinct. For example, if one wants to intersect two plane curves, the curves are not allowed to lie in distinct planes, even if one might consider the planes to be mathematically equivalent.

The basic creation function takes two arguments. The first is the intended base ring, the second the intended dimension. In the code fragment below we make an affine plane (of dimension 2). The x, y delineated by diamond brackets determine the names of the coordinates. This syntax is just the same as that for analogous structures such as polynomial rings.

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

One can retrieve the characteristic data related to a particular plane as in the following examples.

```
> BaseRing(A);
Rational Field
> Dimension(A);
2
> CoordinateRing(A);
Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
```

The variables x and y are named on A, but are really elements of the coordinate ring of A. Thus, `A.1`, the first variable of A which in this case is x, is synonymous with `CoordinateRing(A).1`. Higher dimensional affine ambients are precisely analogous, the difference being that there are more variables.

114.1.2 Curves

Algebraic curves are schemes of dimension one. That is, they are described by the vanishing of a set of polynomials or an ideal in the coordinate ring of their ambient space A and this set of zeros has algebraic dimension 1.

Plane curves are described by the vanishing of a single polynomial equation $f = 0$ in some plane A. In other ambients of dimension d, at least $d-1$ polynomials are needed and usually more (when $d-1$ suffice, we refer to the curve as a global complete intersection).

The polynomials or ideal must belong to the coordinate ring of A and be homogeneous when A is projective. In practical terms, this means that one should usually write the polynomial using the variables of A. When using most of the creation functions below, one must be explicit about the ambient in which the curve lies.

Given some affine plane A with coordinates u, v, define a curve C as the zero locus of a polynomial f in u, v.

```
> A<u,v> := AffineSpace(FiniteField(32003),2);
> f := u^2 - v^5;
> C := Curve(A,f);
> C;
Curve over GF(32003) defined by u^2 + 32002*v^5
```

As we see next, the curve C remains in the ambient space A in which it was constructed. It will always lie in that particular space.

```
> AmbientSpace(C);
Affine space of dimension 2 with coordinates u,v
> AmbientSpace(C) eq A;
true
```

```
> BaseField(C);
Finite field of size 32003
```

Magma can solve systems of polynomial equations over a given field (or even over a domain using resultants). In particular, it can find the singular points of C defined over the current base field.

```
> SingularPoints(C);
{ (0,0) }
> HasSingularPointsOverExtension(C);
false
```

The function `HasSingularPointsOverExtension()` above returns `false` if and only if there are no additional singular points defined over some extension of the base field of C. (It calculates the radical of the Jacobian algebra to check whether the singularities over the current base field account for the whole algebra.) So the origin really is the only singularity of C defined over this field. There are functions which help one find the singularities of C over an extension if they exist.

114.1.3 Projective Closure

Magma maintains a close connection between a curve and its projective closure. Here we illustrate some of the nice results of this.

In this example we define an affine curve in coordinates x, y and take its projective closure. For clarity, we name the homogeneous coordinates on projective space X, Y, Z. These names are really maintained by the projective space containing D even though they appear to have been created with D. Any other curve in this projective space will be expressed in terms of these variables. Names are not automatically given to the projective space. In this example the choice is made at the first opportunity, but it can be made or changed at any time.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,(y^2 - x^3)^2 - y*x^6);
> D<X,Y,Z> := ProjectiveClosure(C);
> D;
Curve over Rational Field defined by
X^6*Y - X^6*Z + 2*X^3*Y^2*Z^2 - Y^4*Z^3
```

Conversely one can retrieve the affine patches of a projective curve. The standard patches are usually the ones of interest, although others can be recovered. The first line below checks that the first patch really is C. Again, variable names are not automatically determined for curves lying in spaces that have not already been created. As seen below, the only result of not assigning names to variables is that the printing is a little unreadable: the first coordinate function is referred to as `$.1` and so on.

```
> AffinePatch(D,1) eq C;
true
> C2 := AffinePatch(D,2);
> C2;
Curve defined by -$.1^6*$.2 + $.1^6 + 2*$.1^3*$.2^2 - $.2^3
```

We name the coordinate functions and check that C_2 really is a patch of D.

```
> A2<u,v> := AmbientSpace(C2);
> C2;
Curve defined by -u^6*v + u^6 + 2*u^3*v^2 - v^3
> ProjectiveClosure(C2);
Curve defined by -X^6*Y + X^6*Z - 2*X^3*Y^2*Z^2 + Y^4*Z^3
```

114.1.4 Points

Points have already arisen as the singular points of a curve. In this chapter we always think of points as being a sequence of coordinates that satisfies the equations of a curve rather than as a zero-dimensional scheme. While the coordinates of a point often lie in the base ring of the curve, they may lie in any extension of that base ring. Technically, the parent of points is not considered to be the curve at all, but instead a *point set.* Point sets are characterised by two pieces of information. The first piece of information is some scheme, in this case a curve C. The second is some algebra over the base ring. (Note that while we say *algebra* here to emphasise the mathematical point, in Magma one uses any ring which admits a map from the base ring rather than an explicit Magma algebra type as the second piece of information.) In other words, given an extension L of the base ring k of a curve C, there is a set, $C(L)$, of points of C with coordinates in L. You can consider the point set $C(L)$ literally to be the set of points of C with coordinates in L, although this set does not actually compute or list all its elements since there are often infinitely many.

Thus points can be thought of as being the closest thing to the notion of point that one uses colloquially: "Is the curve C singular at the point p?", for example, can be translated as `IsSingular(C,p)`.

Creating points is easy. For sequences of coordinates of the base ring of a curve, the expression `C ! [1,2]`, for example, creates the point $(1, 2)$ on the scheme C (assuming that coordinate sequences of length 2 are appropriate for C and that $(1, 2)$ satisfies the equations of C). If the coordinates are in some extension, or if you really want a particular point set as parent, you must be explicit about the point set and write `C(L) ! [1,2]`.

Points may belong to the point sets either of a curve or of the ambient plane of the curve. When a point and a curve are both arguments to a function then this difference isn't visible — in the background the function will check that the point really does lie on the curve if that is what is mathematically required — although being careful about where points lie is good practice. Often one is allowed to omit the curve argument in such functions, in which case one must be clear that the point really does lie exactly where one wants it to lie: `IsNonsingular(p)` is an example of function that is rather susceptible to returning confusing results if one isn't careful about where the point lies.

The `!` operator is Magma's usual coercion operator. It can be used in a variety of situations where one might hope to make natural reinterpretations of objects. Here it is used to reinterpret a sequence of numbers as the coordinates of a point in a plane or a curve.

```
> A<x,y> := AffineSpace(FiniteField(23),2);
> p := A ! [1,2];
> p;
(1, 2)
> q := Origin(A);
> q;
(0, 0)
```

So now it is possible to analyse particular points of a curve.

```
> C := Curve(A,x^2 + 2*x*y^2 - y^5);
> C;
Curve over GF(23) defined by
x^2 + 2*x*y^2 + 22*y^5
> p in C;
true (1, 2)
> IsSingular(C,p);
false
> TangentLine(C,p);
Curve over GF(23) defined by
10*x + 20*y + 19
```

Notice that the statement `p in C` is evaluated in a generous way: p is really in a point set of the plane but it happens to satisfy the equation of the curve and that is what is checked. There are two ways to force the parent of the point to be a point set of C: either use the coercion operator or test it with the standard `IsCoercible()` function. The tangent line to C at p is interpreted as a linear curve embedded in the same plane as C and intersecting it at p. These tangent lines are only defined for plane curves C and are themselves plane curves. In particular, functions which can take a curve as argument can take this tangent line as argument. The tangent line is given the name T in the discussion below and used as an argument in a function which takes two curves and a point. Recall that the tangent line to a curve C at a nonsingular point p is the unique line having intersection number at least 2 with C at p.

```
> T := $1;
> IntersectionNumber(C,T,p);
2
```

114.1.5 Choosing Coordinates

One way to express a curve C after a change of coordinates is to make a map — an automorphism of the ambient space — which realises the change of coordinates and then compute the image of C under this map.

Here is a hands-on analysis of a singularity on a curve which involves changes of coordinates. The example is again a plane curve and involves several functions still only available for such: `TangentCone` and `Blowup`. The singularity is first moved to the origin where coordinates are chosen so that the tangent directions lie along the coordinate axes.

```
> A<x,y> := AffineSpace(GF(11,2),2);
> C := Curve(A,-x^6 + x*y^2 - 2*x*y + x + (y - 1)^3);
> SingularPoints(C);
{ (0, 1) }
> p := Representative(SingularPoints(C));
> f := Translation(A,p);
> C1 := f(C);
> q := Origin(A);
> TangentCone(C1,q);
Curve over GF(11^2) defined by x*y^2 + y^3
> g := Automorphism(A,y);
> C2 := g(C1);
> TangentCone(C2,q);
Curve over GF(11^2) defined by x*y^2
```

The singularity is now in a suitable form for study. The following could be the first steps in an analysis. It can be skipped over by anyone not familiar with the language; otherwise see Chapter 46.

```
> D,E := Blowup(C2);
> IsSingular(D,q),IsSingular(E,q);
true false
> TangentCone(D,q);
Curve over GF(11^2) defined by y^2
> Faces(NewtonPolygon(D));
[ <2, 3, 6> ]
```

So the singularity is almost resolved. One more blowup of the cusp at the origin of D will make a resolution.

114.1.6 Function Fields and Divisors

If C is an integral curve then the field of fractions of its coordinate ring (or the homogeneous part of degree 0 in the projective case), is known as the *function field* of C. The function field allows us to conveniently perform many different computations with the curve. The function fields of an affine curve and its projective closure are isomorphic and are identified in Magma (with the projective version).

In fact, there are two Magma function fields associated to the curve, both abstractly isomorphic to its field of rational functions. What we refer to as the function field here has type `FunFldFracSch` and is associated with more general schemes. Additional information may be found in Chapter 112. This provides the basic user interface for curve functions when they are explicitly needed. For the most part, functions apply to the curve directly with the function field being used in the background. The second function field is an algebraic function field (see Chapter 42). This provides much of the deeper functionality associated to the curve but lies even further in the background and most users should never need to access it directly.

Divisors — loosely speaking, formal sums of points of a curve — are an important part of the technology having many substantial applications. Any nonzero rational function determines a divisor: take the formal sum of zeros of the function on the curve (counted with multiplicity) minus the poles of the function. Divisors of this form are called principal divisors. Conversely, divisors arising in this way determine the function which defined them up to a scalar multiple.

There are two important groups in which divisors are often considered: the divisor group in which addition is simply coefficient-wise, and the divisor class group which is the divisor group modulo the subgroup of principal divisors. In the case of elliptic curves, the class group provides a formal setup in which to interpret the group law.

If you want to see it explicitly, the function field of a curve may be accessed.

```
> k := FiniteField(17);
> P<x,y,z> := ProjectiveSpace(k,2);
> E := Curve(P, y^2*z - x^3 - 4*x*z^2);
> E1<u,v> := AffinePatch(E,1);
> F<a,b> := FunctionField(E);
> F;
Function Field of Curve over GF(17) defined by
16*x^3 + 13*x*z^2 + y^2*z
> Genus(E);
1
```

But often you don't need the function field in your hands since the function you want can be called directly with the curve as argument.

Divisors are constructed by referring to the curve on which they should lie together with some characteristic data for them.

```
> Div := DivisorGroup(E);
> Div;
Group of divisors of Curve over GF(17) defined by
16*x^3 + 13*x*z^2 + y^2*z
> p := E ! [0,0,1];
> L := TangentLine(E,p);
> D := Divisor(E,L);
> D;
Divisor of Curve over GF(17) defined by
16*x^3 + 13*x*z^2 + y^2*z
> Decomposition(D);
[
    <Place at (0 : 0 : 1), 2>
    <Place at (0 : 1 : 0), 1>,
]
```

A little explanation is required. Firstly, the divisor D constructed here is really the divisor of the rational function with zero along L and pole along the line at infinity. Secondly, the basic printing of D is not so helpful: the point is to ensure that potentially lengthy calculations are avoided so it is not immediately printed in 'factorised' form. Next, once factorised, the divisor refers to places of the curve. Since the curve could be singular and divisor computations are done on the nonsingular model, the language of places is used. Note that when printing a place, a point corresponding to the place is shown. Of course, this point does not uniquely characterise the place. If p is a singular point of a curve it is possible to have unequal places that both display p as their support.

In this case, the curve is nonsingular so everything is above board: D is literally the divisor $2p_1 + p_2$ where p_1 is the prime divisor at the point $(0 : 0 : 1)$ and p_2 is the prime divisor at the point $(0 : 1 : 0)$. Now, after defining these prime divisors, we define a new divisor of degree 0.

```
> p1 := Divisor(p);
> p2 := Divisor(E![0,1,0]);
> D2 := D - 3*p1;
> Decomposition(D2);
[
    <Place at (0 : 0 : 1), -1>
    <Place at (0 : 1 : 0), 1>,
]
> Degree(D2);
0
```

The natural question to ask of a divisor of degree 0 is whether or not it is principal.

```
> IsPrincipal(D2);
false
> IsPrincipal(2*D2);
true 1/a
```

So D_2 is not principal but two times D_2 is principal. Moreover, the rational function $1/a$ defines $2 \times D_2$. (This corresponds to the rational function z/x on P.)

Now we look at the class group of E. This function requires that E be defined over a finite field. All the operations above apply to a curve defined over any field.

```
> Cl, _, phi := ClassGroup(E);
> Cl;
Abelian Group isomorphic to Z/4 + Z/4 + Z
Defined on 3 generators
Relations:
    4*Cl.1 = 0
    4*Cl.2 = 0
```

```
> phi;
Mapping from: DivCrv: D to GrpAb: Cl given by a rule
> phi(D2);
2*Cl.2
```

114.2 Ambient Spaces

In this section we show how to create various spaces that can be used as ambient spaces for curves. They can be specified in a number of different ways. Typically, different constructions of such spaces will be taken to be different objects, even if they are defined over the same ring. Names for the coordinates can be defined by using the diamond bracket notation in the same way as for polynomial rings.

The discussion here is rather brief, giving just enough functions to create some basic ambients, their functions and points in them. Consult Chapter 112 for more constructors and functions.

```
AffineSpace(k,n)
AffinePlane(k)
```

Create affine n-dimensional space (resp. the 2-dimensional affine plane) over the ring k.

```
ProjectiveSpace(k,n)
ProjectivePlane(k)
```

Create n-dimensional projective space (resp. the 2-dimensional projective plane) over the ring k.

```
DirectProduct(A,B)
```

If A and B are both one-dimensional projective spaces (defined using the intrinsic `ProjectiveSpace(k,1)` for example) this forms the product $A \times B$ and also returns a sequence containing the two projection maps.

```
RuledSurface(k,n)
RuledSurface(k,a,b)
```

The rational ruled surface over the ring k which has a curve of self-intersection $-n$ or $\pm(a-b)$. The integer arguments must all be nonnegative. It has four variables, the ratio of the first two defining the structure map to $\mathbf{P}^1$, the second two being homogeneous coordinates on the $\mathbf{P}^1$ fibres of this map.

`CoordinateRing(A)`

The coordinate ring of the ambient space A. This is a multivariate polynomial ring over the base ring of A. The number of variables possessed by the ring will depend upon the space. If A is an affine n-space, the function will return a ring with n variables; an ordinary or weighted projective n-space will result in $n+1$ variables; a ruled surface will result in four variables. The gradings that are implicit on various spaces will not be reflected by the polynomial ring. Indeed, at present, there is no way to impose two different gradings on a polynomial ring in Magma.

`FunctionField(A)`

Return the function field of the ambient space A. This is a field isomorphic to the field of fractions of the coordinate ring of A. Its generators can be assigned names in the usual way and, typically, one writes elements of this function field in terms of those generators. Polynomials of the coordinate ring of A can be coerced into this function field.

`A ! [a,...]`

`A(L) ! [a,...]`

For elements $a, \dots$ in the base ring of the ambient space (or any other scheme) A the expression `A ! [a,...]` creates the set-theoretic point, eg, (a, b) in the affine plane case, $(a : b : c)$ in the projective plane case, or $(a : b : c : d)$ in the product or ruled surface case. If L is an extension ring of the base ring of A then the expression `A(L) ! [a,...]` creates the point with coordinates $(a, \dots)$ where these coordinates are elements of L (or the base ring of A).

`Origin(A)`

The point $(0, 0, \dots, 0)$ of the affine space A.

`Coordinates(p)`

`p[i]`

The complete sequence of base ring elements corresponding to the coordinates of the point p or the ith coordinate or p alone.

Example H114E1

In this example we make some points of an affine plane A. The first exhibits the constructor which creates points in the point set of the base ring of A. The second creates a point in some extension of the base ring of A.

```
> k := FiniteField(2);
> A := AffineSpace(k,3);
> p := A ! [1,2,3];
> p;
(1, 0, 1)
> L<w> := ext< k | 2 >;
> q := A(L) ! [1,2,w];
> q[3];
w
```

114.3 Algebraic Curves

A general curve C (type `Crv`) is defined by the vanishing of a finite number of polynomials $f_1, \ldots, f_n$ or a polynomial ideal I in a general ambient space. As a scheme, this must have dimension 1.

A plane curve C (type `CrvPln`) is defined by the vanishing of a single polynomial f in one of the available ambient planes:

$$C : (f_1 = f_2 = \ldots = f_n = 0) \subset A.$$

The polynomials or ideal must lie in the coordinate ring of A. The notation C for a curve and f or $f_1, \ldots, f_n$ for its defining equation(s) will be maintained. The coefficient ring of the parent of polynomials will be denoted k. Irrespective of type, the ambient space will be denoted A.

114.3.1 Creation

In this section the most basic methods of creating a curve are presented. For specialised types — conics, elliptic curves, hyperelliptic curves — there are additional functions documented in the corresponding chapters. Curves may also be created implicitly, such as when they arise as the images of maps.

`Curve(A,f)`

`Nonsingular`	BoolElt	*Default :* `false`
`Reduced`	BoolElt	*Default :* `false`
`Irreducible`	BoolElt	*Default :* `false`
`GeometricallyIrreducible`	BoolElt	*Default :* `false`
`Saturated`	BoolElt	*Default :* `false`

Create the plane curve $f = 0$ in the ambient plane A where f is a polynomial in the coordinate ring of A.

`Curve(A,I)`

`Nonsingular`	BoolElt	*Default :* `false`
`Reduced`	BoolElt	*Default :* `false`
`Irreducible`	BoolElt	*Default :* `false`
`GeometricallyIrreducible`	BoolElt	*Default :* `false`
`Saturated`	BoolElt	*Default :* `false`

Create the curve in the ambient space A determined by the ideal I of the coordinate ring of A. An error results if the result

`Curve(X,S)`

`Nonsingular`	BoolElt	*Default :* `false`
`Reduced`	BoolElt	*Default :* `false`
`Irreducible`	BoolElt	*Default :* `false`
`GeometricallyIrreducible`	BoolElt	*Default :* `false`
`Saturated`	BoolElt	*Default :* `false`

Create the curve defined by the sequence S in the ambient space of X, where S is a sequence of polynomials in the coordinate ring. Here X can be any scheme, not necessarily an ambient space itelf. An error results if the result is not actually a 1-dimensional scheme.

Note: An important special case is when A is an affine or a projective space of dimension 1 and S is empty. This gives the affine or projective line as a curve - as a scheme it is just the ambient space A. Alternatively, the constructor `Curve(A)` described below may be used. Note that the initial construction of A never returns it as a `Crv` type, even though it is 1-dimensional. This is for internal technical reasons – a `Crv` cannot be considered as an ambient space by Magma. is not a 1-dimensional scheme.

`IsCurve(X)`

Returns `true` if and only if X is a one-dimensional scheme.

`Curve(X)`

The smallest scheme in the inclusion chain above the scheme X which is a curve. If X is a curve (ie 1-dimensional) then X will be returned as a `Crv` type. If X has been created as a subscheme of a curve then this curve will be returned.

```
Line(C,p,q)
Line(P,S)
```

The line through the distinct points p, q on the curve C, or the points of S as a subscheme of the projective space P if they are collinear. If the points are points of a curve rather than the ambient space, the line will be interpreted as the tangent line in the case that the points are equal.

```
Conic(P,S)
```

Given a projective plane P and a set S of points in P, this function returns the conic P through the points of the set S if such a conic exists and is unique. The traditional setup corresponds to the case where S is a set of 5 points in general position, that is, no three of them are collinear. If the resulting conic curve is nonsingular, then it will be returned as a special type. See Chapter 119 for details of the special functions that apply in that case.

```
Union(C,D)
```

Create the union of the curves C and D. The result will usually be non-irreducible, so although it will be interpreted as a curve, most of the advanced functions below will not apply to it.

114.3.2 Base Change

Let A be some ambient space in Magma. For example, think of A as being the affine plane. Let k be its base ring and R_A its coordinate ring. If $m : k \to L$ is a map of rings (a coercion map, for instance) then there is a new ambient space denoted A_L and called the *base change of A to L* which has coordinate ring R_A but with coefficients L instead of k. (Mathematically, one simply tensors R_A with L over k. In Magma the equivalent function at the level of polynomial rings is `ChangeRing`.) There is a base change function described below which takes A and L (or the map $k \to L$) as arguments and creates this new space A_L. Note that there is a map from the coordinate ring of A to that of A_L determined by the map m.

This operation is called *base extension* since one often thinks of the map m as being an extension of fields. Of course, the map m could be many other things. One key example where the name *extension* is a little unusual would be when m is the map from the integers to some finite field.

Now let X be a scheme in Magma. Thus X is defined by some polynomials $f_1, \ldots, f_r$ on some ambient space A. Given a ring map $k \to L$ there is a base change operation for X which returns the *base change of X to L*, denoted X_L. This is done by first making the base change of A to L and then using the map from the coordinate ring of A to that of A_L to translate the polynomials f_i into polynomials defined on A_L. These polynomials can then be used to define a scheme in A_L. It is this resulting scheme which is the base change of X to L.

If one has a number of curves in the same ambient space and wants to base change them all at the same time, a little care is required. The function which takes a curve and a map of rings as argument will create a new ambient space each time so should be avoided. A better approach is to apply base change to the ambient space and then invoke the base change function which takes the curve and the desired new ambient space as argument. (This latter base change function appears to be different to the other. In fact it is not. We described base change above as a function of maps of rings. Of course, there is a natural extension to maps of schemes. With that extension, this final base change intrinsic really is base change with respect to map of ambient spaces.)

```
BaseChange(C, K)
```

The base change of the curve C to the new base ring K. This is only possible if elements of the current base ring of C can be coerced automatically into K. The resulting curve will lie in a newly created plane (see the example below).

```
BaseChange(C, m)
```

The base change of the curve C by the map of base rings m. The resulting curve will lie in a newly created plane.

```
BaseChange(C, A)
BaseChange(C, A, m)
```

The base change of the curve C to a curve in the new ambient space A. The space A must be of the same type as the ambient of C and its base ring must either admit coercion from the base ring of C or have the map m between the two explicitly given.

```
BaseChange(C, n)
```

The base change of C, where the base ring of C is a finite field to the finite field which is a degree n extension of the base field of C.

Example H114E2

We give an example of a singular curve, over the rationals, whose singular points are only defined over the field extension given by adjoining a square root of -1.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,y^2 - (x^2+1)^3);
> SingularPoints(C);
{}
> HasSingularPointsOverExtension(C);
true
```

Here we assume that the user knows which extension to move to. The first method of finding the points is to search in the particular point set as follows.

```
> Qi<i> := QuadraticField(-1);
> SingularPoints(C,Qi);
```

```
{ (i, 0), (-i, 0) }
```

The second method is to create a new curve by base change and to search the base ring point set for that curve. For a single calculation this method is rather clumsy, but if further computation were to take place at these points it might be preferable.

```
> B<u,v> := BaseChange(A,Qi);
> Ci := BaseChange(C,B);
> SingularPoints(Ci);
{ (i, 0), (-i, 0) }
```

114.3.3 Basic Attributes

The first few functions below recover data from the ambient space of the curve (and could equally well be applied to the ambient space). Any curves lying in the same ambient space will return identical results when evaluated in these functions. The remaining functions recover data about the equation defining the curve.

The coordinate ring of a curve is described here, but its function field is discussed much later in Section 114.8.

`AmbientSpace(C)`

The ambient space containing the curve C.

`BaseRing(C)`

`CoefficientRing(C)`

`BaseField(C)`

The base ring of the curve C. This is recovered as the base ring of the ambient plane. The third function will report an error if the base ring is not a field.

`DefiningPolynomial(C)`

The defining polynomial of the plane curve C.

`DefiningIdeal(C)`

The defining ideal of the curve C, as an ideal in the coordinate ring of its ambient space.

`CoordinateRing(C)`

The coordinate ring of the curve C. Even creating this requires the use of Gröbner basis techniques.

`Degree(C)`

The degree of the of the curve C which must be defined in an ordinary projective ambient space.

```
JacobianIdeal(C)
```

The ideal of partial derivatives of the defining polynomials of the curve C.

```
JacobianMatrix(C)
```

The matrix of partial derivatives of the defining polynomials of the curve C.

```
HessianMatrix(C)
```

The symmetric matrix of second partial derivatives of the defining polynomial of the plane curve C.

Example H114E3

In this example we start by creating a plane curve C and check that its ideal really is principal. We have chosen an example which is in Weierstrass form.

```
> A<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(A,z*y^2 - x^3 - x*z^2 - z^3);
> IsNonsingular(C);
true
> DefiningIdeal(C);
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Basis:
[
    -x^3 - x*z^2 + y^2*z - z^3
]
> IsPrincipal($1);
true x^3 + x*z^2 - y^2*z + z^3
```

Next we compute the determinant of the Hessian matrix of C. That is a polynomial which we use to create another curve D. The intersection of C and D are the points of inflection, or *flexes*, of C. Over an algebraic closure there will be nine of these, but we only see one — the family "flex at infinity" — over the rationals.

```
> M := HessianMatrix(C);
> Determinant(M);
24*x^2*z + 24*x*y^2 + 72*x*z^2 - 8*z^3
> D := Curve(A,Determinant(M));
> IntersectionPoints(C,D);
{ (0 : 1 : 0) }
```

114.3.4 Basic Invariants

```
IsReduced(C)
```

Returns `true` if and only if the ideal defining the curve C is reduced.

```
IsIrreducible(C)
```

Returns `true` if and only if the curve C is irreducible (as a scheme).

```
IsSingular(C)
```

Returns `true` if and only if the curve C contains at least one singularity over an algebraic closure of its base field.

```
IsNonsingular(C)
```

Returns `true` if and only if the curve C has no singularities over an algebraic closure of its base field.

114.3.5 Random Curves

This section described several functions for the generation of random curves of given degree and/or genus over finite fields and the rationals. The implementations follow [ST02].

```
RandomNodalCurve(d, g, P)
```

`RandomBound` RngIntElt *Default :* 9

Generates a random plane curve in the projective plane P of degree d and genus g with only nodes as singularities. The genus g must satisfy $g \leq (d-1)(d-2)/2$ and then the number of nodes will be $(d-1)(d-2)/2 - g$. These nodes are chosen as a random set of points in P. At the same time g must be $\geq 1 + (d(d-6)/3)$ to guarantee a non-empty linear system for a general set of nodes. For $0 \leq g \leq 10$, a good choice to obtain a general curve of genus g is to take $d = g + 2 - [g/3]$ (see [ST02]).

The base field may be a finite field or $\mathbf{Q}$. Over $\mathbf{Q}$, the construction uses polynomials which will have integer coefficients randomly chosen in the range $[-r \ldots r]$, where r is the value of `RandomBound`. Over a finite field, `RandomBound` is ignored.

```
IsNodalCurve(C)
```

Given a plane curve C, this function returns `true` if either C is non-singular or C only has nodes as singularities.

```
RandomOrdinaryPlaneCurve(d, S, P)
```

`Adjoint`	BoolElt	*Default :* `true`
`Proof`	BoolElt	*Default :* `true`
`RandomBound`	RngIntElt	*Default :* 9

Generates a random plane curve in the projective plane P of degree d and with ordinary singularities specified by the sequence *sings* as follows: If $S = [s_2, s_3, s_4, \ldots]$ then the curve will have s_2 nodes, s_3 triple points, s_4 ordinary singularities of multiplicity 4, etc. For example $[2, 0, 1]$ specifies 2 nodes and one ordinary quadruple point.

For such a curve to exist we require $\binom{d-1}{2} \geq \sum_i s_i \binom{i+1}{2}$.

If `Proof` is `false` then the full check that the singularities are ordinary is skipped.

If `Adjoint` is `true` then the adjoint ideal, an ideal of the coordinate ring of P, is also computed and returned as a second value. The r-th graded parts of this homogeneous ideal realises the linear system $K + (r - d + 3)H$ on the normalisation of the curve, where K is the canonical divisor and H is the hyperplane section divisor corresponding to the (singular) embedding into P. This can be used to compute various adjoint maps (for example, $r = d - 3$ gives the canonical map) and its computation by this function is more efficient than using the general method of blowing-up in `Adjoints` (this function now also tests for ordinariness and uses the adjoint ideal by default).

The base field can be finite or $\mathbf{Q}$ and `RandomBound` is as before.

```
RandomCurveByGenus(g, K)
```

`RandomBound`	RngIntElt	*Default :* 9

Given a positive integer g and a field K this function generates a random projective curve over K of genus g, for $0 \leq g \leq 13$. When $g \leq 10$, a plane nodal curve is returned as given by the function `RandomNodalCurve` with degree $g + 2 - [g/3]$. For $11 \leq g \leq 13$, a curve in $\mathbf{P}^3$ is returned, computed by syzygy computations as described in [ST02].

The field K must be a finite field or $\mathbf{Q}$. The parameter `RandomBound` applies when K is chosen to be $\mathbf{Q}$. Note that, although $\mathbf{Q}$ is allowed in all cases, for the higher values of g, particularly for $g \geq 11$, the heights of the coefficients of the defining polynomials for the curve produced tend to be very large, even for small values of `RandomBound`.

Example H114E4

```
> SetSeed(1);
> C := RandomCurveByGenus(4, Rationals());
> C;
Curve over Rational Field defined by
x^5 + 34965/512*x*y*z^3 - 59355/512*x*z^4 + y^5 - 16705/48*y^3*z^2 -
    1831885/1536*y^2*z^3 - 1553135/2304*y*z^4 + 655145/4608*z^5
```

```
> Genus(C);
4
> C := RandomCurveByGenus(8, GF(23));
> C;
Curve over GF(23) defined by
17*x^8 + 5*x^7*y + 5*x^7*z + 13*x^6*y^2 + 11*x^6*y*z + 8*x^6*z^2 + 16*x^5*y^3 +
    17*x^5*y^2*z + 22*x^5*y*z^2 + 6*x^5*z^3 + 3*x^4*y^4 + 2*x^4*y^3*z +
    18*x^4*y^2*z^2 + 3*x^4*y*z^3 + 14*x^4*z^4 + 19*x^3*y^5 + 19*x^3*y^4*z +
    19*x^3*y^3*z^2 + 21*x^3*y^2*z^3 + 21*x^3*y*z^4 + 10*x^3*z^5 + 18*x^2*y^6 +
    4*x^2*y^5*z + 9*x^2*y^4*z^2 + 2*x^2*y^3*z^3 + 21*x^2*y^2*z^4 + 22*x^2*y*z^5
    + 6*x^2*z^6 + 14*x*y^7 + 4*x*y^6*z + 17*x*y^5*z^2 + 20*x*y^4*z^3 +
    14*x*y^2*z^5 + 15*x*y*z^6 + 3*x*z^7 + 18*y^8 + 2*y^7*z + 11*y^6*z^2 +
    18*y^4*z^4 + 18*y^3*z^5 + 5*y^2*z^6 + 22*y*z^7 + 2*z^8
> Genus(C);
8
> C := RandomCurveByGenus(12, GF(23));
> Ambient(C);
Projective Space of dimension 3
Variables : x, y, z, t
> Degree(C); Genus(C);
12
12
```

114.3.6 Ordinary Plane Curves

The term ordinary plane curve refers to a curve in the projective plane all of whose singularities are ordinary. This means that a singularity of multiplicity $m \geq 2$ has m “distinct tangent directions” – the equation of the curve expanded in local coordinates at the singularity begins with a binary form of degree m which splits into m distinct linear factors over the algebraic closure of the ground field.

A significant property of such curves is that all of their singularities are resolved by a single blow-up. Their adjoint linear systems/adjoint ideal can be computed in a more direct fashion than for more general plane curves. These linear systems give important projective maps such as the canonical map for curves of genus at least 2 and embeddings as rational normal curves for curves of genus 0.

This section contains some functions relating to ordinary curves and, in particular, to nodal curves, all of whose singularities are ordinary of multiplicity 2 (nodes). The list of functions is likely to be extended in future versions of Magma.

```
HasOnlyOrdinarySingularities(C)
```

`Adjoint` BoolElt *Default :* `true`

Given a plane curve C this function returns `true` in C has only ordinary singularities. If that is the case it also returns the maximum of the multiplicities of the singularities of C (1 means that C is non-singular). Further, if C is ordinary and if `Adjoint` is `true`, then the (saturated) adjoint ideal is also computed and returned as the third value.

```
HasOnlyOrdinarySingularitiesMonteCarlo(C)
```

Given a plane curve C defined over $\mathbf{Q}$ perform a Monte Carlo test for ordinariness. This will generally be faster than the intrinsic `OnlyOrdinarySingularities`. The function does not compute the adjoint ideal. Five primes are chosen for which the mod p reduction of C is still a curve and which has Jacobian ideal of the same degree as that of C. The five reductions are tested for ordinary singularities. If all pass, then `true` is returned. Otherwise `false` is returned. If `false` is returned, then C is definitely not ordinary. If it succeeds, then C is very likely to be ordinary but this is not 100% guaranteed.

```
AdjointIdeal(C)
```

Returns the (saturated) adjoint ideal of an ordinary plane curve C. If C is not ordinary then an error results.

```
AdjointIdealForNodalCurve(C)
AdjointLinearSystemForNodalCurve(C, d)
```

Given a plane curve C that is *assumed* to be nodal, these are slightly faster intrinsics for computing the adjoint ideal and adjoint linear system, respectively. The first function returns the adjoint ideal I and the second returns the degree d adjoint linear system, which is the linear subsystem of the complete plane linear system of degree d given by the degree d graded part of I.

```
AdjointLinearSystemFromIdeal(I, d)
```

Given an ideal I and a positive integer d, this function returns the degree d adjoint linear system for a plane curve whose (saturated) adjoint ideal is I.

```
CanonicalLinearSystemFromIdeal(I, d)
```

Given an ideal I and a positive integer d, this function returns the canonical linear system for a plane curve of degree d whose (saturated) adjoint ideal is I. This is the same as intrinsic `AdjointLinearSystemFromIdeal` with I and $d-3$ as arguments. It will be empty if the curve has genus 0.

`CanonicalLinearSystem(C)`

`AdjointLinearSystem(C)`

`Adjoints(C,d)`

Given a plane curve C the first two functions return the canonical linear system for any plane curve C and the third gives the general degree d adjoint linear system. If C is ordinary, then the functions compute the adjoint ideal and takes its graded piece as above. If not, they have to work out in detail the graph of the full resolution of singularities of C, which can take some time.

Example H114E5

In this example, we generate a random ordinary plane curve of degree 7 with 3 nodes and one ordinary singularity of order 4. We use its adjoint ideal to get the canonical map and compute its canonical image in $\mathbf{P}^5$. The computation of the canonical map this way is generally faster and gives a much simpler map description than the computation for general curves using the function field machinery.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C, I := RandomPlaneCurve(7,[3,0,1],P : RandomBound := 2);
> C;
Curve over Rational Field defined by
x^7 + x^5*y*z + x^4*y^3 + x^3*y^2*z^2 - 708*x^3*z^4 - 3401/18*x^2*y^2*z^3 +
    3274/9*x^2*y*z^4 + 29054/9*x^2*z^5 - 5861/15552*x*y^6 - 5279/432*x*y^5*z -
    71851/1296*x*y^4*z^2 + 173479/486*x*y^3*z^3 + 87337/108*x*y^2*z^4 -
    294685/81*x*y*z^5 - 662891/243*x*z^6 + 8117/15552*y^7 + 5543/972*y^6*z +
    41419/1296*y^5*z^2 - 17087/243*y^4*z^3 - 646577/972*y^3*z^4 +
    77026/81*y^2*z^5 + 423635/243*y*z^6 + 156920/243*z^7
> Genus(C);
6
> //check with OnlyOrdinarySingularities function
> boo,d,I1 := HasOnlyOrdinarySingularities(C);
> boo; d;
true
4
> I eq I1;
true
> // polynomials for canonical map come from the degree d-3=4
> // graded piece of the adjoint ideal.
> can_pols := AdjointLinearSystemFromIdeal(I, 4);
> Sections(can_pols);
[
    x^4 - 24*x^2*z^2 - 83/36*x*y^2*z + 83/9*x*y*z^2 + 493/9*x*z^3 - 167/432*y^4
        + 145/108*y^3*z + 35/6*y^2*z^2 - 731/27*y*z^3 - 587/27*z^4,
    x^3*y - 12*x^2*z^2 - 73/18*x*y^2*z + 38/9*x*y*z^2 + 286/9*x*z^3 - 55/216*y^4
        + 65/54*y^3*z + 7*y^2*z^2 - 494/27*y*z^3 - 350/27*z^4,
    x^3*z - 6*x^2*z^2 - 5/18*x*y^2*z + 10/9*x*y*z^2 + 98/9*x*z^3 - 7/108*y^4 +
        5/27*y^3*z + y^2*z^2 - 112/27*y*z^3 - 112/27*z^4,
    x^2*y^2 - 4*x^2*z^2 - 11/3*x*y^2*z - 4/3*x*y*z^2 + 52/3*x*z^3 - 5/36*y^4 -
        5/9*y^3*z + 10*y^2*z^2 - 116/9*y*z^3 - 68/9*z^4,
    x^2*y*z - 2*x^2*z^2 - 2/3*x*y^2*z - 4/3*x*y*z^2 + 16/3*x*z^3 - 1/18*y^4 +
        5/18*y^3*z + y^2*z^2 - 14/9*y*z^3 - 20/9*z^4,
    x*y^3 - 12*x*y*z^2 + 16*x*z^3 - 1/2*y^4 - 2*y^3*z + 12*y^2*z^2 - 8*y*z^3 -
        8*z^4
]
> X := CanonicalImage(C, Sections(can_pols));
> X;
Curve over Rational Field defined by
x[1]*x[4] - x[2]^2 + 5/36*x[2]*x[6] + 4*x[3]^2 + 11/3*x[3]*x[4] - 68/3*x[3]*x[5]
    - 68/9*x[3]*x[6] - 95/216*x[4]*x[6] + 140/3*x[5]^2 + 2*x[5]*x[6] +
    131/324*x[6]^2,
x[1]*x[5] - x[2]*x[3] + 1/18*x[2]*x[6] + 2*x[3]^2 + 2/3*x[3]*x[4] -
    14/3*x[3]*x[5] - 5/9*x[3]*x[6] - 1/27*x[4]*x[6] + 7/6*x[5]^2 +
    1/24*x[5]*x[6] + 2/81*x[6]^2,
-x[1]*x[6] + x[2]*x[4] - 1/2*x[2]*x[6] - 8*x[3]*x[5] + 5/3*x[3]*x[6] -
    1/9*x[4]*x[6] + 14*x[5]^2 + 4*x[5]*x[6] + 43/216*x[6]^2,
x[2]*x[5] - x[3]*x[4] + 2*x[3]*x[5] + 2/3*x[3]*x[6] + 1/18*x[4]*x[6] - 7*x[5]^2
    - 1/4*x[5]*x[6] - 1/27*x[6]^2,
-x[2]*x[6] + x[4]^2 - 1/2*x[4]*x[6] - 4*x[5]^2 + 16/3*x[5]*x[6] + 1/36*x[6]^2,
-x[3]*x[6] + x[4]*x[5] + 2*x[5]^2 + 1/6*x[5]*x[6] + 1/18*x[6]^2,
x[1]^3 - 10*x[1]^2*x[3] + x[1]*x[2]*x[3] + 52*x[1]*x[3]^2 + 95/216*x[2]^3 +
    15/2*x[2]^2*x[3] + 337/432*x[2]^2*x[4] - 491/18*x[2]*x[3]^2 -
    425/36*x[2]*x[3]*x[4] - 71/1296*x[2]*x[4]^2 + 676/27*x[3]^3 +
    3907/108*x[3]^2*x[4] - 506/27*x[3]^2*x[5] + 341/36*x[3]*x[4]^2 -
    529/36*x[3]*x[4]*x[5] + 2525/162*x[3]*x[5]^2 - 103/15552*x[4]^3 -
    18385/1944*x[4]^2*x[5] + 4751/11664*x[4]^2*x[6] - 44317/1296*x[4]*x[5]^2 +
    317441/46656*x[4]*x[5]*x[6] - 3677/139968*x[4]*x[6]^2 - 11905/162*x[5]^3 -
    52453/11664*x[5]^2*x[6] - 1113761/279936*x[5]*x[6]^2 +
    813889/1679616*x[6]^3,
x[1]^2*x[2] - 2*x[1]^2*x[3] - 8*x[1]*x[2]*x[3] + 16*x[1]*x[3]^2 - 31/36*x[2]^3 +
    29/3*x[2]^2*x[3] + 113/72*x[2]^2*x[4] + 29/3*x[2]*x[3]^2 -
    5/9*x[2]*x[3]*x[4] + 5/54*x[2]*x[4]^2 - 460/9*x[3]^3 - 577/18*x[3]^2*x[4] +
    1688/9*x[3]^2*x[5] - 1391/108*x[3]*x[4]^2 + 2257/18*x[3]*x[4]*x[5] -
    1717/9*x[3]*x[5]^2 - 437/648*x[4]^3 + 15593/648*x[4]^2*x[5] +
    13321/15552*x[4]^2*x[6] - 10073/108*x[4]*x[5]^2 - 93221/7776*x[4]*x[5]*x[6]
    + 93403/93312*x[4]*x[6]^2 + 1582/9*x[5]^3 - 11059/1296*x[5]^2*x[6] -
    193157/46656*x[5]*x[6]^2 + 363041/559872*x[6]^3,
x[1]*x[2]^2 - 4*x[1]*x[2]*x[3] + 4*x[1]*x[3]^2 - 7/6*x[2]^3 + x[2]^2*x[3] +
    1/2*x[2]^2*x[4] + 10*x[2]*x[3]^2 + 26/3*x[2]*x[3]*x[4] + 10/9*x[2]*x[4]^2 -
    44/3*x[3]^3 - 36*x[3]^2*x[4] + 200/3*x[3]^2*x[5] - 301/18*x[3]*x[4]^2 +
    331/3*x[3]*x[4]*x[5] - 538/3*x[3]*x[5]^2 - 83/54*x[4]^3 +
    1109/54*x[4]^2*x[5] + 13/8*x[4]^2*x[6] - 1117/12*x[4]*x[5]^2 -
    4051/648*x[4]*x[5]*x[6] - 25/1296*x[4]*x[6]^2 + 2236/9*x[5]^3 +
    14917/648*x[5]^2*x[6] - 1351/162*x[5]*x[6]^2 + 45109/46656*x[6]^3
```

114.4 Local Geometry

Here we discuss some basic functions providing analysis of a point p lying on a curve C. Firstly we describe how to create points on curves and their basic access functions. One should also refer to the comments in Section 112.7 of the general schemes chapter about the point sets of point arguments of these functions where there is a fuller discussion of point sets.

Most functions usually have two arguments, a curve and a point on that curve. In fact, the point need not actually be in a point set of the curve since coercion will be attempted if it is not. Moreover, the curve argument is not strictly necessary either, since if the point does lie in a point set of the curve, it can be recovered automatically. So these functions also work with the curve argument omitted. However, omitting the curve argument should be thought of merely as a convenient shorthand and should be used with care — it is very easy to use a point from some other space for which the function still makes sense but returns a misleading answer.

114.4.1 Creation of Points on Curves

Points of a curve C, and indeed points of any scheme in Magma, lie in point sets associated to C rather than C itself. Each point set is the parent of points whose coordinates lie in a particular extension ring of the base ring of the curve. Thus, if k is the base ring of the curve C, points whose coordinates lie in k are elements of the "base ring point set" denoted $C(k)$. If L is an extension ring of k (in the sense of admitting coercion from k or being the codomain of a ring homomorphism from k) then points with coordinates in L lie in the point set $C(L)$.

Here we give the basic point creation methods and access functions. For more information, consult the discussion of points and point sets in Section 112.7 of Chapter 112 on schemes.

```
C ! [a,...]
```

For a sequence of elements $a, \ldots$ of the base ring of C, this creates the point of C with coordinates $(a, \ldots)$. The parent of the resulting point is the base point set of the curve C rather than C itself.

```
C(L) ! [a,...]
```

For a sequence of elements $a, \ldots$ of the extension ring L of the base ring of C, this function creates the point of C with coordinates $(a, \ldots)$. The parent of the resulting point is the point set $C(L)$ of the curve C rather than C itself. The phrase 'extension ring' here means that either L admits automatic coercion from the base ring of C, or that L is the codomain of a ring homomorphism from that base ring.

```
Curve(p)
```

The smallest scheme in the inclusion chain above the scheme on which the point p lies which is a curve. If p lies on a curve then the curve will be returned.

```
Curve(P)
```

The smallest scheme in the inclusion chain above the scheme P is a point set of which is a curve. If P is a point set of a curve then this curve will be returned.

```
Coordinates(p)
```

The sequence of ring elements corresponding to the coordinates of the point p.

```
p[i]
Coordinate(p,i)
```

The ith coordinate of the point p.

```
p eq q
```

Returns `true` if and only if the two points p and q lie in schemes contained in a common ambient space, have coordinates that can be compared (either by lying in the same ring, or by an automatic coercion) and these coordinates are equal.

```
FormalPoint(P)
```

Given a non–singular point P in $C(K)$, where C is a curve and K is some extension of the field of definition of C, returns a point in `C(LaurentSeriesRing(K))`, such that specializing the variable to 0 yields P.

114.4.2 Operations at a Point

Most of the functions in this section report an error if p does not lie on C. Functions having arguments C, p allow the omission of C as long as the parent of p is a point set of C. A few of the functions apply only to plane curves.

```
p in C
S in C
```

Returns `true` if and only if the point p or the sequence of coordinates S lies on the curve C. That is, return `true` if and only if the coordinates of p satisfy the equation of C.

```
IsNonsingular(C,p)
```

Returns `true` if and only if p is a nonsingular point of the curve C.

```
IsSingular(C,p)
```

Returns `true` if and only if the point p is a singular point on the curve C.

```
IsInflectionPoint(C,p)
IsFlex(C,p)
```

Returns `true` if and only if the point p is a flex of the plane curve C. An error is reported if p is a singular point of C. The second return value is the order of the flex, that is, the local intersection number at p of C with its tangent line at p.

```
TangentLine(p)
TangentLine(C,p)
```

The tangent line to the curve C at the point p embedded as a curve in the same space; an error if p is a singular point of C.

```
TangentCone(C,p)
```

The tangent cone to the curve C at the point p embedded in the same ambient space.

```
IsTangent(C,D,p)
```

Returns `true` if and only if the plane curves C and D are nonsingular and tangent at the point p.

114.4.3 Singularity Analysis

These functions report an error if p is not a singular point of C. Again, the arguments can be abbreviated to just the point if care is taken about its parent.

```
Multiplicity(C,p)
```

The multiplicity of the curve C at the point p.

```
IsDoublePoint(C,p)
```

Returns `true` if and only if the point p is a double point of the curve C.

```
IsOrdinarySingularity(C,p)
```

Returns `true` if and only if the point p is a singular point of the curve C with reduced tangent cone.

```
IsNode(C,p)
```

Returns `true` if and only if the point p is an ordinary double point of the curve C.

```
IsCusp(C,p)
```

Returns `true` if and only if the point p is a nonordinary double point of the curve C.

```
IsAnalyticallyIrreducible(C,p)
```

Returns `true` if and only if the plane curve C has exactly one place at the point p, or equivalently if the resolution of singularities is injective above p.

Example H114E6

Each of the two curves in this example has a double point at the origin. One of these is a node and one is a cusp.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,x^2-y^3);
> p := Origin(A);
> IsCusp(C,p);
true
> IsDoublePoint(C,p);
true
> IsReduced(TangentCone(C,p));
false
> D := Curve(A,x^2 - y^3 - y^2);
> IsAnalyticallyIrreducible(D,p);
false
> IsNode(D,p);
true
```

114.4.4 Resolution of Singularities

Again, all functions in this section only apply to plane curves.

`Blowup(C)`

Given the affine plane curve C, return the two affine plane curves lying on the standard patches of the blowup of the affine plane at the origin. Note that the two curves returned are the *birational* transforms of C on the blowup patches. The patches are contained in the same affine space as the curve itself. If C does not contain the origin this returns an error message.

`Blowup(C,M)`

This returns the weighted blowup of the plane curve C at the origin defined by the 2×2 matrix of integers M. Again, the birational transform of C is returned inside the ambient plane of C. An error is reported if M does not have determinant ± 1.

Example H114E7

It often happens that one can replace a string of ordinary blowups used to resolve a curve singularity by a single weighted blowup.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,y^2 - x^7);
> f := map< A -> A | [x^2*y,x^7*y^3] >;
> C @@ f;
Curve over Rational Field defined by
x^14*y^7 - x^14*y^6
```

```
> M := Matrix(2,[2,1,7,3]);
> Blowup(C,M);
Curve over Rational Field defined by
-y + 1
14 6
```

The blowup function takes the total pullback as the underlying map and then removes all copies of the x and y axes. The pair of numbers displayed in the final line is the multiplicity of these factors in the total pullback. The curve returned is the birational pullback of C on some patch of a rational surface arising by a number of blowups above the origin of A. It is clearly nonsingular — it's linear! — so this map resolves the singularity at the origin of C.

In fact, Magma has machinery for interpreting strings of blowups in terms of a graph, the *resolution graph*.

```
> ResolutionGraph(C);
The resolution graph on the Digraph
Vertex  Neighbours
1 ([ -2, 7, 4, 0 ])     2 ;
2 ([ -1, 14, 8, 1 ])    3 ;
3 ([ -3, 6, 3, 0 ])     4 ;
4 ([ -2, 4, 2, 0 ])     5 ;
5 ([ -2, 2, 1, 0 ])     ;
```

Consult Chapter 115 for the full interpretation of this graph. Briefly, one should see this as representing a chain of five blowups which resolve the curve. Each vertex of the graph corresponds to one of the exceptional curves coming from these blowups. The curve extracted by the weighted blowup we saw above corresponds to vertex number 2. Indeed, we can see the multiplicity 14 in the total pullback as the second entry of the labelling sequence. (The multiplicity 6 which we saw above is the corresponding entry in exceptional curve 3.) The fourth entry of that sequence, 1, reports that the birational transform of C to the blownup surface intersects the exceptional curve with multiplicity 1. This is the only nonzero fourth entry of any vertex label, so we conclude that there is exactly one place above the singularity at the origin. This can be confirmed (more quickly!) by the divisor machinery which will be discussed in Section 114.9.

```
> Places(C ! Origin(A));
[
    Place at (0 : 0 : 1)
]
> Degree($1[1]);
1
```

114.4.5 Log Canonical Thresholds

For background on log canonical singularities, see for example [Kollár 1997, Singularities of pairs] or [Kollár 1998, Birational geometry of algebraic varieties].

Let V be a variety with at worst log canonical singularities, P a point on V and D an effective $\mathbf{Q}$-Cartier divisor on V. Then the *log canonical threshold* (lct) of the log pair (V, D) at P is the number

$$\mathrm{lct}_P(V, D) = \sup\{\lambda \in \mathbf{Q} \mid (V, \lambda D) \text{ is log canonical at } P\} \in \mathbf{Q} \cup \{+\infty\}.$$

We can also consider the lct of D along the whole of V:

$$\begin{aligned}\mathrm{lct}(V, D) &= \inf\{\mathrm{lct}_P(V, D) \mid P \in V\}\\ &= \sup\{\lambda \in \mathbf{Q} \mid (V, \lambda D) \text{ is log canonical}\}.\end{aligned}$$

```
LogCanonicalThreshold(C)
```

The log canonical threshold of the curve C computed at its singular k-points, where k is the base field of C.

```
LogCanonicalThresholdAtOrigin(C)
```

The local log canonical threshold of the affine curve C computed at the origin.

```
LogCanonicalThreshold(C, P)
```

The local log canonical threshold of the curve C computed at the point P.

```
LogCanonicalThresholdOverExtension(C)
```

The log canonical threshold of the curve C computed at all singular points including those defined over some base field extension.

Example H114E8

Consider a cubic curve C on the projective plane, then the singularities of C resemble one of the following examples: a smooth curve, e.g.,

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> A := Curve(P2,x^3-y^2*z-3*x*z^2);
> IsNonsingular(A);
true
```

a curve with ordinary double points (i.e. nodes), e.g.,

```
> B := Curve(P2,x^3-y^2*z-3*x*z^2+2*z^3);
> IsNodalCurve(B);
true
```

a curve with one cuspidal point, e.g.,

```
> C := Curve(P2,x^3-y^2*z);
> #SingularPoints(C) eq 1;
```

```
true
> IsCusp(C,SingularPoints(C)[1]);
true
```

a conic and a line that are tangent, e.g.,

```
> D := Curve(P2,(x^2+(y-z)^2-z^2)*y);
> #PrimeComponents(D) eq 2;
true
> TangentCone(PrimeComponents(D)[1],P2![0,0,1]) eq PrimeComponents(D)[2];
true
```

three lines intersecting at one (Eckardt) point, e.g.

```
> E := Curve(P2,x*y*(x-y));
> IsOrdinarySingularity(E,P2![0,0,1]);
true
> Multiplicity(E,P2![0,0,1]);
3
```

a curve whose support consists of two lines, e.g.,

```
> F := Curve(P2,x^2*y);
> IsReduced(F);
false
> #SingularPoints(ReducedSubscheme(F)) eq 1;
true
> IsNodalCurve(Curve(ReducedSubscheme(F)));
true
```

or a curve whose support consists of three lines, e.g.,

```
> G := Curve(P2,x^3);
> IsReduced(G);
false
> IsNonsingular(ReducedSubscheme(G));
true
```

It is known that a curve is log canonical whenever its singularities are at worst nodal, thus $\mathrm{lct}(P^2, A) = \mathrm{lct}(P^2, B) = 1$. For the remaining reduced curves we can resolve their singularities and calculate their discrepancies to find their log canonical thresholds.

```
> curves := [* A,B,C,D,E,F,G *];
> [LogCanonicalThreshold(curve) : curve in curves];
```

It follows that $\mathrm{lct}(P^2, -K_{P^2}) \leq \frac{1}{3}$. In fact, it is not hard to see that equality holds.

Example H114E9

Here we exhibit a curve C over the rationals, **Q**, that has singularities defined over a splitting field, k, where $\mathrm{lct}(C)$ (over k) $<$ $\mathrm{lct}(C)$ (over **Q**). We take a curve C in the projective plane $P2$ with one ordinary double point and two triple point singularities. Such a curve can be obtained by calling:

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := RandomPlaneCurve(6,[1,2],P2);
```

For this example we use the fixed curve C defined below.

```
> f := x*y^5 + y^6 + x^5*z + x^2*y^3*z + 2095/3402*y^5*z + x^4*z^2 -
>      6244382419/8614788*x^3*y*z^2 -
>      28401292681/8614788*x^2*y^2*z^2 -
>      89017753225/25844364*x*y^3*z^2 -
>      243243649115/232599276*y^4*z^2 -
>      2798099890675/70354102*x^3*z^3 -
>      22754590185549/281416408*x^2*y*z^3 -
>      7190675316787/140708204*x*y^2*z^3 -
>      75304687887883/7598243016*y^3*z^3 +
>      17778098933653/140708204*x^2*z^4 +
>      6098447759659/35177051*x*y*z^4 +
>      24308031251845/422124612*y^2*z^4 -
>      4694415764252/35177051*x*z^5 -
>      77497995284599/844249224*y*z^5 +
>      6592790982389/140708204*z^6;
> C := Curve(P2,f);
> IsSingular(C);
true
> LogCanonicalThreshold(C);
1
> IsNodalCurve(C);
false
```

Thus C must have singularities defined over some field extension.

```
> HasSingularPointsOverExtension(C);
true
> LogCanonicalThresholdOverExtension(C);
2/3
```

114.4.6 Local Intersection Theory

The main function here for a single point uses a standard Euclidean algorithm to calculate local intersection numbers at points where two plane curves meet. It was taken from unpublished lecture notes of Franz Winkler, "Introduction to Commutative Algebra and Algebraic Geometry"; the same algorithm is in Fulton's book [Ful69]. These numbers are also called *intersection multiplicities* in the literature. In the following sections there are functions for finding the intersection points of two curves.

There is now (to be added to an early patch release of V2.19) a variant that uses an algorithm of Jan Hilmar and Chris Smyth described in [HS10]. This computes all intersection points of the two curves as a set of Galois conjugacy classes and their intersection numbers in a single computation. The implementation, adapted into Magma from code contributed by Chris Smyth, returns the sequence of points along with the corresponding local intersection multiplicities.

```
IsIntersection(C,D,p)
```

Returns `true` if and only if the point p lies on both curves C and D.

```
IsTransverse(C,D,p)
```

Returns `true` if and only if the point p is a nonsingular point of both plane curves C and D and the curves have distinct tangents there.

```
IntersectionNumber(C,D,p)
```

The local intersection number $I_p(C, D)$ of the plane curves C and D at the point p. This reports an error if C or D have a common component at p.

```
IntersectionNumbers(C,D)
IntersectionNumbers(F,G)
```

These intrinsics will be added in an early patch release of V2.19. They use the algorithm of Hilmar and Smyth to compute in one go a list of all intersection places along with the corresponding local intersection multiplicities of two projective plane curves C and D defined over the rationals $\mathbf{Q}$. Here, intersection place means a point in $\mathbf{P}^2(K)$, where $\mathbf{P}^2$ is the ambient of C and D and K is a finite extension of $\mathbf{Q}$, which represents a Galois conjugacy class of ($[K : \mathbf{Q}]$) points in the intersection of C and D.

The first intrinsic takes C and D as arguments (which must have no common irreducible component) and returns the result as a list of pairs $\langle p, m\rangle$, where p is an element of a pointset $\mathbf{P}^2(K)$ giving a place in the intersection and m is the corresponding local intersection multiplicity.

The second intrinsic avoids the use of pointsets. It takes two homogeneous polynomials F and G which are relatively prime and lie in the same multivariate polynomial ring $P = \mathbf{Q}[x, y, z]$. They represent two plane curves in $Proj(P)$ and the result is a list of intersection places of these curves with intersection multiplicity. The elements of the list are again pairs $\langle p, m\rangle$, but here p is represented as in Hilmar and Smyth's paper [HS10], p is a list of three elements of one of three types:

i) $[*1, 0, 0*]$. Represents the plane projective point with these homogeneous coordinates.

ii) $[*f(x), 1, 0*]$ with $f(x)$ an irreducible polynomial in $\mathbf{Q}[x]$. Represents the conjugate points with homogeneous coordinates $[\alpha, 1, 0]$ where α ranges over the roots of f.

iii) $[*h(x, y), g(y), 1*]$ with $g(y)$ an irreducible polynomial in $\mathbf{Q}[y]$ and $h(x, y)$ a polynomial in $\mathbf{Q}[x, y]$ whose image in $(\mathbf{Q}[y]/(g(y)))[x]$ is irreducible. Represents the conjugate points with homogeneous coordinates $[\gamma, \beta, 1]$ where β ranges over the roots of g and, for each β, γ ranges over the roots of $h(x, \beta)$.

Example H114E10

The local intersection of two curves at a point where they share a common tangent is calculated. If the curves did not share a tangent, the intersection would be the product of multiplicities which it is not in this case.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A, y^2 - x^5);
> D := Curve(A, y - x^2);
> p := Origin(A);
> IntersectionNumber(C,D,p);
4
> Multiplicity(C,p) * Multiplicity(D,p);
2
```

These intersection numbers are often defined to be the length of a particular affine algebra. (See [Har77] Chapter I, Exercise 5.4.) Below it is checked that this definition produces the same result in this case. Note that the algebra is not localised at p so the length calculated is the sum of intersection numbers at all intersection points. At the end one sees that the discrepancy of 1 is accounted for by a single transverse intersection away from the origin.

```
> RA := CoordinateRing(A);
> I := ideal< RA | DefiningPolynomial(C), DefiningPolynomial(D) >;
> Dimension(RA/I);
5
> IP := IntersectionPoints(C,D);
> IP;
{@ (0, 0), (1, 1) @}
> IsTransverse(C,D,IP[1]);
false
> IsTransverse(C,D,IP[2]);
true
```

114.5 Global Geometry

In this section functions which determine global properties of curves such as their genus and whether their equation has a particular form are presented.

114.5.1 Genus and Singularities

```
Genus(C)
GeometricGenus(C)
```

The topological genus of the curve C. More precisely, this is the arithmetic genus of the projective normalisation $\tilde{C}$, which is unique up to k-isomorphism, where k is the basefield of C. C must be an integral curve (reduced and irreducible as a scheme).

Note that, if k is not a perfect field, $\tilde{C}$ may have singularities over an inseparable extension field of k (in technical terms, $\tilde{C}$ is a non-singular scheme, but it may not be k-smooth), in which case the genus of C may drop after some (inseparable) basefield extensions.

```
ArithmeticGenus(C)
```

The arithmetic genus of the curve C or its projective closure if C is affine. In the case of a plane projective curve of degree d, this number is just $(d-1)(d-2)/2$.

This is really the arithmetic genus of (projective) scheme C and not of its normalisation.

```
NumberOfPunctures(C)
```

The number of punctures of the affine plane curve C over an algebraic closure of its ground field, that is, the number of points supporting its reduced scheme at infinity. This is just the reduced degree of the polynomial of C at infinity.

```
SingularPoints(C)
```

The singular points of the curve C which are defined over the base field of C.

```
HasSingularPointsOverExtension(C)
```

Returns `false` if and only if the scheme of singularities of the curve C has support defined over the base field of C. This function requires that C be reduced.

```
Flexes(C)
InflectionPoints(C)
```

For a plane curve C, this returns the subscheme of C defined by the vanishing of the determinant of the Hessian matrix. This contains the “flex points” of C, which by definition are the nonsingular points at which the tangent line intersects C with multiplicity at least 3.

```
C eq D
```

Returns `true` if and only if the curves C and D are defined by identical ideals in the same ambient space. (For plane curves, this simply compares defining polynomials of the two curves up to a factor so Gröbner basis calculations are avoided.)

```
IsSubscheme(C,D)
```

Returns `true` if and only if the curve C is contained (scheme-theoretically) in the curve D.

Example H114E11

We take a plane affine cubic C with a single cusp and non-singular at infinity. Here the projective normalisation of C is isomorphic to the projective line with genus 0, although the arithmetic genus of C is 1.

```
> A<x,y> := AffineSpace(GF(3),2);
> C := Curve(A,y^2 - x^3 - 1);
> Genus(C);
0
> ArithmeticGenus(C);
1
```

Now we consider a similar cubic over the non-perfect field $K = k(t)$ with a single cuspidal singularity defined over an inseparable cubic extension. Now C is a normal and non-singular scheme (but non-smooth), which only loses its normality after an inseparable basefield extension. Here both the genus and arithmetic genus are 1.

```
> K<t> := RationalFunctionField(GF(3));
> A<x,y> := AffineSpace(K,2);
> C := Curve(A,y^2 - x^3 - t);
> Genus(C);
1
```

114.5.2 Projective Closure and Affine Patches

In Magma, any affine space has a unique projective closure. This may be assigned different variable names just like any projective space. The projective closure functions applied to affine curves will return projective curves in the projective closure of the affine ambient. Conversely, a projective space has standard affine patches. These will also appear as the ambient spaces of the standard affine patches of a projective curve.

```
ProjectiveClosure(A)
```

The projective space that is the projective closure of the ambient A. Unless A is already expressed as a particular patch on some projective space, this is the standard closure defined by the homogenisation of the coordinate ring of A with a new coordinate and unit weights.

`ProjectiveClosure(C)`

The closure of the curve C in the projective closure of its ambient affine space.

Example H114E12

Since the closure of the ambient space is unique, the ambient space of the closure of curves lying in a common affine space is independent of how it is constructed. Here is an odd example but one that occurs in practice when curves and spaces are passed between functions: the functions `ProjectiveClosure()` and `AmbientSpace()` commute!

```
> A<a,b> := AffineSpace(GF(5),2);
> C := Curve(A,a^3 - b^4);
> AmbientSpace(ProjectiveClosure(C)) eq ProjectiveClosure(AmbientSpace(C));
true
```

`LineAtInfinity(A)`

The line which is the complement of the affine plane A embedded in the projective closure of A.

`PointsAtInfinity(C)`

The set of points at infinity defined over the base field of the curve C. The number of these points can also be recovered by the `NumberOfPunctures()` function in the plane case.

`AffinePatch(C,i)`

The i-th affine patch of the projective curve C. For ordinary projective space, the first patch is the one centred on the point $(0 : 0 : \ldots : 0 : 1)$, the second at the point $(0 : 0 : \ldots : 1 : 0)$ and so on.

Example H114E13

Usually one looks at the first affine patch of a curve. If the curve is described, as below, using homogeneous coordinates x,y,z then this is often realised by “setting $z = 1$”. Note that we have to assign names to the coordinates explicitly on the affine patches if we want them.

```
> P<x,y,z> := ProjectiveSpace(GF(11),2);
> C := Curve(P,x^3*z^2 - y^5);
> AffinePatch(C,1);
Curve over GF(11) defined by 10*$.1^3 + $.2^5
> C1<u,v> := AffinePatch(C,1);
> C1;
Curve over GF(11) defined by 10*u^3 + v^5
> SingularPoints(C);
```

```
{ (1 : 0 : 0), (0 : 0 : 1) }
```

One can also look at other patches. Indeed, sometimes it is necessary. In this example, the curve C has an interesting singularity "at infinity", the point $(1 : 0 : 0)$. If we want to view it on an affine curve then we must take one of the other patches.

```
> C3<Y,Z> := AffinePatch(C,3);
> C3;
Curve over GF(11) defined by Y^5 + 10*Z^2
> IsSingular(C3 ! [0,0]);
true
```

Both affine curves $C1$ and $C3$ have the projective curve C as their projective closure.

```
> ProjectiveClosure(C1) eq ProjectiveClosure(C3);
true
```

114.5.3 Special Forms of Curves

The functions in this section check whether a curve is written in a particular normal form, and also whether it belongs to one of the more specialised families of curve.

`IsEllipticWeierstrass(C)`

Returns `true` if the curve C is nonsingular plane curve of genus 1 in Weierstrass form. This tests the coefficients of the polynomial of C. The conditions guarantee a flex at the point $(0 : 1 : 0)$ either on C or on its projective closure. These are precisely the conditions required by the linear equivalence algorithms for divisors in a later section.

`IsHyperellipticWeierstrass(C)`

Returns `true` if the curve C is a hyperelliptic curve in plane Weierstrass form. The conditions chosen are that the (a) first affine patch be nonsingular, (b) the point $(0 : 1 : 0)$ is the only point at infinity and has tangent cone supported at the line at infinity and (c), the projection of C away from that point has degree 2.

`EllipticCurve(C)`

`EllipticCurve(C,p)`

`EllipticCurve(C,p)`

See the description of `EllipticCurve` in Chapter 120.

`IsHyperelliptic(C)`

`Eqn` BoolElt *Default :* `true`

`IsGeometricallyHyperelliptic(C)`

`Map` BoolElt *Default :* `true`

Verbose `IsHyp` *Maximum :* 2

The second function determines whether the curve C is a hyperelliptic curve over the algebraic closure of its base field. If so and if `Map` is `true`, a plane conic or the projective line and a degree 2 map from C to it (all defined over the base field) are returned. The map is to the line if the genus of C is even and to a conic if the genus is odd.

The first function determines whether the curve C is hyperelliptic over its base field K (ie has a degree 2 map to the projective line defined over K). If so, and if the `Eqn` parameter is `true`, it also returns a hyperelliptic Weierstrass model H over K and an isomorphic scheme map from C to H.

Here, hyperelliptic entails genus ≥ 2.

The basic method in both cases is to find the image of C under the canonical map (using functions to be described later) and to check if this is of arithmetic genus zero. If so, this image curve (which is rational normal) is mapped down to a plane conic or the line by repeated adjunction maps. For the second function, the final equation is determined by differential computations in the function field of C once the explicit map to the projective line, which gives the base x function, has been determined.

Example H114E14

```
> P<a,b,c,d,e,f> := ProjectiveSpace(Rationals(),5);
> C := Curve(P,[
> a^2 + a*c - c*e + 3*d*e + 2*d*f - 2*e^2 - 2*e*f - f^2,
> a*c - b^2,
> a*d - b*c,
> a*e - c^2,
> a*e - b*d,
> b*e - c*d,
> c*e - d^2
> ] );
> boo,hy,mp := IsHyperelliptic(C);
> boo;
true
> hy;
Hyperelliptic Curve defined by y^2 = x^8 + x^6 + x - 1 over Rational Field
> mp;
Mapping from: Sch: C to CrvHyp: hy
with equations :
c*e - d^2 + d*f
```

```
-d*f + e*f + f^2
e*f
```

114.6 Maps and Curves

114.6.1 Elementary Maps

The first group of functions create selfmaps of the affine plane. Such a map f can be used to move a curve around the plane simply by applying it to the curve. See Chapter 112 on schemes for more details about maps.

```
IdentityAutomorphism(A)
```

```
Translation(A,p)
```

```
FlipCoordinates(A)
```

```
Automorphism(A,q)
```

These are the basic automorphisms of the affine plane A taking (x, y) to (x, y), $(x - a, y - b)$, (y, x) and $(x + q(y), y)$ respectively, where p is the point (a, b) and q is a polynomial on A involving y only.

```
TranslationToInfinity(C,p)
```

The image of C under the change of coordinates which translates p to the point $(0 : 1 : 0)$ in the projective plane and makes the tangent line there equal to the line at infinity. An error is reported if p is a singular point of C. The change of coordinates map is given so that other curves can be mapped by the same change of coordinates.

Example H114E15

In this example we show how one could begin to work out a Weierstrass equation for a Fermat cubic. First we define that cubic curve C in the projective plane and choose a point p on C.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^3 + y^3 + z^3);
> p := C ! [1,-1,0];
> IsFlex(C,p);
true 3
```

The point we have chosen is a flex — the second return value of 3 is the local intersection number of the curve C with its tangent line at p. We use the intrinsic `TranslationToInfinity` to make an automorphism of P which takes the point p to the point $(0 : 1 : 0)$ and takes the curve C to a curve which has tangent line $z = 0$ at the image of p.

```
> C1,phi := TranslationToInfinity(C,p);
> phi(p);
(0 : 1 : 0)
```

```
> C1;
Curve over Rational Field defined by
x^3 + 3*y^2*z - 3*y*z^2 + z^3
```

This is almost in Weierstrass form already. It is a pleasant exercise to make coordinate changes which "absorb" some of the coefficients. Alternatively, one can use the intrinsic `EllipticCurve` to perform the entire transformation in one step.

`EvaluateByPowerSeries(m, P)`

Given a map $m : C \to D$, and a nonsingular point P on C, where C is a curve, return $m(P)$, evaluating $m(P)$ using a power series expansion if necessary. This allows a rational map on C to be evaluated at nonsingular base points.

Example H114E16

The following example shows a map evaluated at a point using power series methods.

```
> P2<X,Y,Z>:=ProjectiveSpace(Rationals(),2);
> C:=Curve(P2,X^3+Y^3-2*Z^3);
> D:=Curve(P2,Y^2*Z-X^3+27*Z^3);
> phi:=map<C->D|[-6*X^2-6*X*Z+6*Y^2+6*Y*Z,
>                9*X^2+18*X*Y+18*X*Z+9*Y^2+18*Y*Z+36*Z^2,
>                X^2-2*X*Z-Y^2+2*Y*Z
>               ]>;
> P:=C![-1,1,0];
> P in BaseScheme(phi);
true (-1 : 1 : 0)
> Q:=EvaluateByPowerSeries(phi,P);
> Q;
(3 : 0 : 1)
>  phi(P);
>> phi(P);
      ^
Runtime error in map application: Image of map does not lie in the codomain
> pullbackQ:=Q@@phi;
> pullbackQ;
Scheme over Rational Field defined by
-9*X^2 + 9*Y^2,
9*X^2 + 18*X*Y + 18*X*Z + 9*Y^2 + 18*Y*Z + 36*Z^2,
X^3 + Y^3 - 2*Z^3
> IsSubscheme(BaseScheme(phi), pullbackQ);
true
> P in pullbackQ;
true (-1 : 1 : 0)
> Degree(BaseScheme(phi))+1 eq Degree(pullbackQ);
true
```

114.6.2 Maps Induced by Morphisms

Given a non-constant map $\phi : D \to C$ between curves, there are several induced maps between the function fields of C and D and the divisor groups $\mathrm{Div}(C)$ and $\mathrm{Div}(D)$. We refer to the contravariant maps ϕ^* as *Pullback*s and to the covariant maps ϕ_*, corresponding to the Norm between the function fields, as *Pushforward*s. Divisor groups and other function field related items are discussed in Section 114.8.

```
Degree(m)
```

Returns the degree of a non-constant dominant map m between curves.

```
RamificationDivisor(m)
```

Returns the ramification divisor of a non-constant dominant map m between irreducible curves.

```
Pullback(phi, X)
```

Given a map $\phi : D \to C$ between curves and a function, differential, place or divisor X on C, this function returns the pullback of X along ϕ.

```
Pushforward(phi, X)
```

Given a map $\phi : D \to C$ between curves and a function, place or divisor X on C, this function returns the pushforward of X along ϕ. In older versions, the function applied to a place used to only work with the image of the point (or cluster) below the place for speed and would give an error when ϕ was undefined there. Now, if this is true, the function reverts to working entirely with places and should never fail.

Example H114E17

As an illustration of these routines, consider the following example

```
> Puvw<u,v,w>:=ProjectiveSpace(Rationals(),2);
> Pxyz<x,y,z>:=ProjectiveSpace(Rationals(),2);
> D:=Curve(Puvw,u^4+v^4-w^4);
> C:=Curve(Pxyz,x^4-y^4+y^2*z^2);
> phiAmb:=map<Puvw->Pxyz|[y*z,z^2,x^2]>;
> phi:=Restriction(phiAmb,D,C);
> KC:=FunctionField(C);
> KD:=FunctionField(D);
> Omega:=BasisOfHolomorphicDifferentials(C)[1];
```

Here we see a holomorphic differential pulls back to holomorphic.

```
> IsEffective(Divisor(Pullback(phi,Omega)));
true
```

Ramification divisors are actually quite easy to compute.

```
> RamificationDivisor(phi) eq
>     Divisor(Pullback(phi,Omega))-Pullback(phi,Divisor(Omega));
true
```

Verifying Riemann-Hurwitz:

```
> 2*Genus(D)-2 eq Degree(phi)*(2*Genus(C)-2)+Degree(RamificationDivisor(phi));
true
```

Pulling back and pushing forward is taking powers on the function field.

```
> f:=KC.1;
> Pushforward(phi,Pullback(phi,f)) eq f^Degree(phi);
true
```

`Divisor` and `Pushforward` commute.

```
> g:=KD.1;
> Divisor(Pushforward(phi,g)) eq Pushforward(phi,Divisor(g));
true
```

114.7 Automorphism Groups of Curves

Magma now (V2.13) contains functionality for computing the automorphism group of a (reduced and irreducible) curve over a variety of base fields. New structures have been added to aid the user in working with these groups. An automorphism group is of type `GrpAutCrv` and there may be several of these associated to a given curve: the full automorphism group or a subgroup generated by a given set of automorphisms. The automorphism group structure acts as a container for a permutation group representation of the group of curve automorphisms along with the map between the representing group and a list of the actual automorphisms stored in a compressed format.

Elements of the group are of type `GrpAutCrvElt`, a subtype of `MapAutSch`. For these, all of the usual scheme-map operations are available. However, a number of these are specially implemented for the new type and there are also several new operations. In particular, composition and powering of elements makes use of the internal group representation for speed and there are functions for the actions on points, places, divisors etc. which are more efficient than those for a general (finite) map between curves. There is also a function to compute the matrix representation of an automorphism group on the space of holomorphic differentials of the curve.

The full automorphism group is computed at function field level. The algebraic function field versions of some of the functions here are described in Section 42.7.2.

It should be stressed that these groups are groups of *birational* automorphisms of a curve C, in the usual way. This means that they are correspond to actual automorphisms (ie, everywhere defined with an everywhere defined inverse) of the unique normalisation of C and the full automorphism group is the full automorphism group of this normalisation. However, if C is singular, then some of these automorphisms may not be defined at particular singular points.

There are also functions for computing isomorphisms between distinct curves.

All functions require C to have a function field, whether the full group of automorphisms is computed or not.

All of the functions computing the full set of automorphisms require the base field to be perfect (characteristic zero or finite).

114.7.1 Group Creation Functions

`AutomorphismGroup(C)`

Given a reduced, irreducible projective curve C, this function returns the full automorphism group of C over its base field. If the genus of C is less than 2, an error results unless the base field is finite. Note that the full automorphism group is cached so is only ever computed once.

`AutomorphismGroup(C,auts)`

With C as above, this function returns the automorphism group of C generated by the sequence of automorphisms *auts*. The sequence *auts* can consist of either scheme automorphisms of C of type `MapAutSch` or `GrpAutCrvElt` elements lying in a previously constructed automorphism group of C. The same restrictions on the genus apply.

`Automorphisms(C)`

`Bound` RngIntElt *Default :* ∞

For C as above, this function computes at most `Bound` automorphisms of C over its base field and returns them as a sequence of scheme maps (type `MapSch`). If `Bound` is `Infinity` or at least the order of the full automorphism group, all automorphisms will be returned.

`IsIsomorphic(C, D)`

Given irreducible curves C and D this function returns `true` is C and D are isomorphic over their common base field. If so, it also returns a scheme map giving an isomorphism between them. The curves C and D must be reduced. Currently the function requires that the curves are not both genus 0 nor both genus 1 unless the base field is finite.

`Isomorphisms(C, D)`

`Bound` RngIntElt *Default :* ∞

Given reduced, irreducible curves C and D this function returns at most `Bound` isomorphisms from C to D over their common base field. These are returned as a sequence of scheme maps. The genus restrictions are as for `IsIsomorphic`.

114.7.2 Automorphisms

```
A . i
```

Let A be a group of automorphisms of curve C and let i be an integer such that $-n \leq i \leq n$, where n is the number of generators of A. This operator returns the i-th generator for A. A negative subscript indicates that the inverse of the generator is to be created. Finally, $A.0$ denotes the identity of A.

Note that if A is an automorphism group that was generated from a list of specified automorphisms, *auts*, then the generators of A will not necessarily be these "user" generators, but will be those coming from the permutation representation of A.

```
Identity(A)
```

```
Id(A)
```

```
A ! 1
```

The identity element of the automorphism group A.

```
A ! f
```

Let A be an automorphism group of a curve C. Given a scheme map f from C to C or an element of some (other) automorphism group of C, this function returns the `GrpAutCrvElt` element of A corresponding to f. An error will result if f is not in A.

```
Order(f)
```

The order of the curve automorphism f.

```
Inverse(f)
```

The inverse of the curve automorphism f.

```
f * g
```

The product of the curve automorphisms f and g in automorphism group A. If f and g are regarded as maps, this function returns their composite: first apply f, then apply g. Note that this composition uses the permutation representation of A and is generally performed faster than the usual scheme map composition.

```
f ^ n
```

The nth power of the curve automorphism f. The integer n may be positive or negative. Again, this uses the permutation representation of A, the parent of f.

```
g eq h
```

Given curve automorphisms g and h belonging to automorphism groups of the same curve C, return `true` if g and h represent the same automorphism, `false` otherwise. Note that g and h do not have to belong to the same automorphism group.

```
g ne h
```

The logical negation of the preceding function.

```
SchemeMap(f)
```

It is sometimes useful for the user to convert a curve automorphism back to an object of plain scheme isomorphism type. This is a convenience function that simply returns the curve automorphism f as a `MapAutSch`.

114.7.3 Automorphism Group Operations

```
Curve(A)
```

The curve of which A is a group of automorphisms.

```
Order(A)
```

The order of A.

```
FactoredOrder(A)
```

The factored order of A.

```
NumberOfGenerators(A)
Ngens(A)
```

The number of generators of A. If A was defined as the group generated by a given sequence of automorphisms, this sequence will not necessarily coincide with the set of generators, which is determined by the internal permutation representation.

```
Generators(A)
```

The generators of A - a small set of `GrpAutCrvElt` elements of A, which generate it as a group - returned in a sequence.

```
PermutationGroup(A)
```

The permutation group of A which gives the abstract group representation of the automorphism group.

```
PermutationRepresentation(A)
```

The permutation group representation of A consisting of the actual permutation group G and an invertible map which takes elements of G to the corresponding curve automorphism (as a `GrpAutCrvElt` element).

```
MatrixRepresentation(A)
```

For an automorphism group A on a curve C of genus at least 2, computes the matrix representation of A acting on the space of holomorphic differentials by pullback. The differentials are represented by row vectors with respect to a basis B and the matrix associated to $g \in A$ gives the pullback action of g by right multiplication of the row vectors. The (finite) matrix group image G is returned along with a map from A to G giving the representation and an enumerated sequence containing the basis of differentials B used.

```
a in A
```

Returns whether a curve automorphism, given as a `GrpAutCrvElt` or `MapSch`, is equal to an automorphism lying in A.

```
A subset B
```

Returns whether A is actually a subgroup of B as automorphism groups of the same curve.

114.7.4 Pullbacks and Pushforwards

For `GrpAutCrvElt` elements, pullbacks and pushforwards (images) of functions, places, divisors and differentials of the curve C are handled more directly than for general curve maps. The functions to perform these operations use the more direct `f(x)` and `x @@ f` syntax and these should be used in preference to the `Pushforward` and `Pullback` functions in Section 114.6.2.

In addition, the computation of images of points on C by a curve automorphism is handled slightly differently. If the point doesn't lie in the original domain of definition of the scheme map giving the automorphism, the image is still computed without extending the map (although for some singular points of the curve model, the image may still not exist for the "birational" automorphism).

```
f(X)
X @ f
```

Given a curve automorphism f of curve C and a point, function, differential, place or divisor X on C, this function returns the image (or pushforward) of X under f.

```
X @@ f
```

Given a curve automorphism f of curve C and a function, differential, place or divisor X on C, this function returns the inverse image (or pullback) of X under f.

Example H114E18

Here are some examples of the above functions applied to the genus 3 Fermat curve in characteristic 0.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^4+y^4+z^4);
> L := Automorphisms(C);
> #L;
24
> // to get all automorphisms, we base change to Q(zeta_8)
> K := CyclotomicField(8);
> C1 := BaseChange(C,K);
> L1 := Automorphisms(C1);
> #L1;
96
> // next, we get the automorphism as a group
> G := AutomorphismGroup(C1);
> g := G!iso<C1 -> C1 | [y,z,x],[z,x,y]>;
> Gp,rep := PermutationRepresentation(G);
> Gp;
Permutation group Gp acting on a set of cardinality 12
Order = 96 = 2^5 * 3
    (2, 4)(3, 5)(6, 8)(7, 10)
    (2, 5, 10, 8, 4, 3, 7, 6)(11, 12)
    (1, 2, 3)(4, 5, 9)(6, 11, 7)(8, 12, 10)
> rep(g);
(1, 8, 4)(2, 9, 6)(3, 10, 11)(5, 7, 12)
> Inverse(rep)(Gp.2);
Mapping from: CrvPln: C1 to CrvPln: C1
with equations :
zeta_8^2*$.1^3*$.3
$.1^3*$.2
$.2^4 + $.3^4
and inverse
zeta_8^2*$.1^3*$.3
-zeta_8^2*$.1^3*$.2
$.2^4 + $.3^4
> $1 eq G.2;
true
```

Example H114E19

In the next example we look at a superelliptic curve over $GF(7^2)$ with isomorphism group a central extension of $PGL_2(\mathbf{F}_7)$ by a cyclic group of order 4.

```
> SetSeed(1);
> k := GF(7^2);
> A<x,y> := AffineSpace(k,2);
> C := ProjectiveClosure(Curve(A,y^4-x^7+x));
> G := AutomorphismGroup(C);
> Gp,rep := PermutationRepresentation(G);
> Gp;
Permutation group G acting on a set of cardinality 192
Order = 1344 = 2^6 * 3 * 7
> [Order(G.i) : i in [1..Ngens(G)]];
[ 7, 24, 2 ]
> Z := Centre(Gp); Z;
Permutation group Z acting on a set of cardinality 192
Order = 4 = 2^2
> (Z.1)@@rep;
Mapping from: CrvPln: C to CrvPln: C
with equations :
$.1
k.1^12*$.2
$.3
and inverse
$.1
k.1^36*$.2
$.3
> Gp1 := quo<Gp|Z>;
> boo := IsIsomorphic(Gp1,PGL(2,GF(7)));
> boo;
true
> // Find the representation on the 176 Weierstrass places.
> // Only need the action of the generators of Gp (or G)
> wpls := WeierstrassPlaces(C);
> wpls_perms := [[Index(wpls,g(w)) : w in wpls]: g in Generators(G)];
> G_wpls := SymmetricGroup(#wpls);
> weier_rep := hom<Gp->G_wpls|[G_wpls!p : p in wpls_perms]>;
> //Check that its faithful
> K := Kernel(weier_rep);
> #K;
1
```

Example H114E20

The next example illustrates the use of `IsIsomorphic` in finding an isomorphism between the well-known plane model of genus 3 curve $X(7)$ and a degree 6 model of it in $\mathbf{P}^3$. The isomorphism between C and D is only computed in one direction, but we can then use `IsInvertible`.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2,x^3*y+y^3*z+z^3*x);
> P3<a,b,c,d> := ProjectiveSpace(Rationals(),3);
> D := Curve(P3,[b^2-a*d,a*b*c+b*d^2+c^3]);
> boo,im := IsIsomorphic(C,D);
> boo;
```

```
true
> im;
Mapping from: CrvPln: C to Crv: D
with equations :
x^2
x*z
y*z
z^2
> _,imi := IsInvertible(im);
> Inverse(imi);
Mapping from: CrvPln: C to Crv: D
with equations :
x^2
x*z
y*z
z^2
and inverse
b
c
d
```

We now compute the automorphism group of C over $GF(11^3)$ where the full set of 168 automorphisms exists (the group is isomorphic to $PSL_2(\mathbf{F}_7)$) and generate some subgroups.

```
> P2<x,y,z> := ProjectiveSpace(GF(11^3),2);
> C := Curve(P2,x^3*y+y^3*z+z^3*x);
> G := AutomorphismGroup(C);
> Order(G);
168
> [Order(g) : g in Generators(G)];
[ 7, 3, 7 ]
> G1 := AutomorphismGroup(C,[G.1,G.2]);
> Order(G1);
21
> PermutationGroup(G1);
Permutation group acting on a set of cardinality 8
Order = 21 = 3 * 7
    (2, 4, 3, 7, 5, 8, 6)
    (3, 5, 4)(6, 8, 7)
> // can also find the normaliser of <G.3> via the
> // permutation rep
> Gp,rep := PermutationRepresentation(G);
> H := Normaliser(Gp,sub<Gp|Gp.3>);
> #H;
21
> Hgens := [g@@rep : g in Generators(H)];
> [Order(g) : g in Hgens];
[ 3, 7 ]
> h := Hgens[1];
> //check directly that h normalises <g> in G
> g := G.3;
> Index([g^i : i in [1..7]],h*g*(h^-1));
4
```

114.7.5 Quotients of Curves

For G an arbitrary group of automorphisms of a curve C of genus >= 2, the main intrinsic in this section computes a model of the quotient curve C/G along with the explicit projection map $C \to C/G$.

```
CurveQuotient(G)
```

G is a group of automorphisms of a curve C/k, which must be of genus g >= 2. The function computes a projective, non-singular model of C/G, the scheme theoretic quotient of C by G. This is returned with the G-invariant projection map from C down to it.

If $k(C)$ is the function field of C then C/G can be thought of as the curve with function field $k(C)^{G^*}$ where G^* is the group of field automorphisms of $k(C)/k$ induced by G under function pull-back.

The implementation utilises Magma's Function Field and Invariant Theory functionality and uses a variety of methods. There is no restriction on the characteristic p and the function works in positive characteristic as long as the following assumption is true:

$$C \to C/G \text{ is tamely ramified when genus}(C/G) > 1$$

This is equivalent to saying that, for all points P on (the non-singular, projective model of) C, the subgroup G_P of G fixing P has order prime to p. In practise, this will only be a problem for very small $p > 0$ when $p|\#G$.

The algorithm used is slightly different depending on g_G, the genus of the quotient curve.

When g_G >= 2 and the quotient is non-hyperelliptic, the canonical image of C/G is computed and this is the model returned. This uses function field functionality only.

The case g_G >= 2 and the quotient is hyperelliptic is similar, only extra work is needed to find the "y-coordinate". Again everything is done purely with function fields. The model returned is a `CrvHyp` Weierstrass model if the quotient is hyperelliptic over k, or a bi-quadratic model in P^3 if it is only geometrically hyperelliptic.

When g_G is 0 or 1, the canonical map methodology used in the above cases fails and we use a combination of Invariant theory and function field methods instead.

For $g_G = 0$, the model returned for C/G is either the projective line P^1 or a conic in P^2. In this case, the function does not automatically search for k-rational points so, if a conic is returned, the quotient may still be isomorphic to P^1 over k.

For $g_G = 1$, the model returned is a projectively normal embedding by quadrics in P^n, $n >= 3$, or a cubic in P^2. Again, the function does not search for k-rational points in order to try to convert the quotient into elliptic curve form.

Example H114E21

We start with our old friend the Klein quartic again - this time over Q. The quotient by an automorphism of order 3 is a genus 1 curve. We find this quotient and produce an isomorphism from this to an elliptic curve using the rational point that is the projection of $(0 : 0 : 1)$.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2,x^3*y+y^3*z+z^3*x);
> phi := iso<C->C|[y,z,x],[z,x,y]>;
> // we will take the quotient by phi
> G := AutomorphismGroup(C,[phi]);
> CG,prj := CurveQuotient(G);
> CG;
Curve over Rational Field defined by
x[1]^2 - 13*x[1]*x[2] + 8*x[1]*x[3] + 10*x[2]*x[3] - 6*x[3]^2 + 15*x[1]*x[4] +
    3*x[2]*x[4] - 6*x[3]*x[4] - 6*x[4]^2,
x[1]^2 - 12*x[1]*x[2] + 3*x[2]^2 + 8*x[1]*x[3] + 12*x[1]*x[4]
> Genus(CG);
1
> // find an minimal elliptic Weierstrass model
> ptCG := prj(C![0,0,1]);
> E1, psi1 := EllipticCurve(CG,ptCG);
> E, psi := MinimalModel(E1);
> prj := Expand(prj * psi1 * psi); // get the composite map C -> E
> E;
Elliptic Curve defined by y^2 + x*y = x^3 - x^2 - 2*x - 1 over Rational Field
> prj;
Mapping from: CrvPln: C to CrvEll: E
with equations :
-2*x^2*y^8 + 2*x*y^8*z - 2*y^9*z - 2*x*y^7*z^2 - 2*x^2*y^5*z^3 + 2*x*y^6*z^3 -
    2*y^7*z^3 + 2*x^2*y^4*z^4 - 2*x^2*y^3*z^5 - 2*y^4*z^6 + 2*y^3*z^7 -
    2*y^2*z^8
4*x^2*y^8 - 2*x*y^9 - 2*x^2*y^7*z - 2*x*y^8*z + 4*y^9*z + 2*x^2*y^6*z^2 +
    4*x*y^7*z^2 - 2*y^8*z^2 + 2*x^2*y^5*z^3 - 4*x*y^6*z^3 + 4*y^7*z^3 -
    4*x^2*y^4*z^4 + 2*x*y^5*z^4 - 2*y^6*z^4 + 4*x^2*y^3*z^5 + 2*y^5*z^5 -
    2*x^2*y^2*z^6 + 2*y^4*z^6 - 4*y^3*z^7 + 4*y^2*z^8 - 2*y*z^9
2*x*y^8*z - 2*x^2*y^5*z^3 - 2*y^4*z^6
> Conductor(E);
49
```

In fact, C is the modular curve $X(7)$ and E is $X_0(49)$. That C/G and E are isogenous elliptic curves also follows from modular form theory.

Example H114E22

As a second example we consider the genus 4 modular curve $X_0(54)$ with the two commuting Atkin-Lehner involutions W_2, W_{27}. We compute the quotients by $< W_2 >$, $< W_{27} >$ and $< W_2, W_{27} >$ which respectively have genera 2,1,0.

```
> P<[x]> := ProjectiveSpace(Rationals(),3);
> X054 := Curve(P,[
>   x[2]*x[3] - x[1]*x[4], 4*x[1]^2*x[2] + 2*x[1]*x[2]^2 +
>       x[2]^3 - x[3]^3 + x[3]^2*x[4] - x[3]*x[4]^2]);
> W2 := iso<X054->X054|[1/2*x[4],-x[3],-x[2],2*x[1]],
>                  [1/2*x[4],-x[3],-x[2],2*x[1]]>;
> W27 := iso<X054->X054|
> [
>     -1/3*x[1] - 1/3*x[2] - 1/3*x[3] - 1/3*x[4],
>     -2/3*x[1] - 2/3*x[2] + 1/3*x[3] + 1/3*x[4],
>     -2/3*x[1] + 1/3*x[2] - 2/3*x[3] + 1/3*x[4],
>     -4/3*x[1] + 2/3*x[2] + 2/3*x[3] - 1/3*x[4]
> ],
> [
>     -1/3*x[1] - 1/3*x[2] - 1/3*x[3] - 1/3*x[4],
>     -2/3*x[1] - 2/3*x[2] + 1/3*x[3] + 1/3*x[4],
>     -2/3*x[1] + 1/3*x[2] - 2/3*x[3] + 1/3*x[4],
>     -4/3*x[1] + 2/3*x[2] + 2/3*x[3] - 1/3*x[4]
> ]>;
> // 1. Quotient by <W2>
> G := AutomorphismGroup(X054,[W2]);
> CG,prj := CurveQuotient(G);
> CG;
Hyperelliptic Curve defined by y^2 + (-x^2 - 1)*y = 387*x^6 + 999*x^5 + 785*x^4
    + 294*x^3 + 58*x^2 + 6*x over Rational Field
> Genus(CG);
2
> // 2. Quotient by <W27>
> G := AutomorphismGroup(X054,[W27]);
> CG,prj := CurveQuotient(G);
> CG;
Curve over Rational Field defined by
1878243840*x[1]*x[2] - 68400*x[2]^2 - 680252400*x[1]*x[3] - 774110424*x[2]*x[3]
    + 246079248*x[3]^2 + 1252162560*x[1]*x[4] - 298086240*x[2]*x[4] -
    834823168*x[3]*x[4] - 2254334400*x[1]*x[5] - 447968988*x[2]*x[5] +
    18*x[3]*x[5] + 936*x[4]*x[5] + 346615560*x[1]*x[6] + 63486504*x[2]*x[6] -
    1620*x[5]*x[6] + 14787*x[6]^2,
-4225630080*x[1]*x[2] + 152304*x[2]^2 + 1530412160*x[1]*x[3] +
    1741572584*x[2]*x[3] - 553624072*x[3]^2 - 2817086720*x[1]*x[4] +
    670624096*x[2]*x[4] + 1878164832*x[3]*x[4] + 5071740600*x[1]*x[5] +
    1007828352*x[2]*x[5] - 1824*x[4]*x[5] - 779805580*x[1]*x[6] -
    142830348*x[2]*x[6] + 4*x[3]*x[6] + 3870*x[5]*x[6] - 33021*x[6]^2,
-14377204800*x[1]*x[2] + 512928*x[2]^2 + 5207031920*x[1]*x[3] +
    5925482424*x[2]*x[3] - 1883642044*x[3]^2 - 9584803200*x[1]*x[4] +
    2281707168*x[2]*x[4] + 6390228448*x[3]*x[4] + 64*x[4]^2 +
    17255963160*x[1]*x[5] + 3429009288*x[2]*x[5] - 5472*x[4]*x[5] -
    2653188900*x[1]*x[6] - 485962956*x[2]*x[6] + 14040*x[5]*x[6] -
    111537*x[6]^2,
762945840*x[1]*x[2] - 27648*x[2]^2 - 276319320*x[1]*x[3] - 314444676*x[2]*x[3] +
    99957860*x[3]^2 + 508630560*x[1]*x[4] - 121082832*x[2]*x[4] -
    339106480*x[3]*x[4] - 915713640*x[1]*x[5] - 181965582*x[2]*x[5] +
    360*x[4]*x[5] + 140795640*x[1]*x[6] + 25788312*x[2]*x[6] - 675*x[5]*x[6] +
    5985*x[6]^2,
559393920*x[1]*x[2] - 18864*x[2]^2 - 202594400*x[1]*x[3] - 230548712*x[2]*x[3] +
    73289896*x[3]^2 + 372929280*x[1]*x[4] - 88775136*x[2]*x[4] -
    248632672*x[3]*x[4] - 671395320*x[1]*x[5] - 133415856*x[2]*x[5] +
    103229820*x[1]*x[6] + 18907956*x[2]*x[6] + 16*x[4]*x[6] - 702*x[5]*x[6] +
    4167*x[6]^2,
488083680*x[1]*x[2] - 16272*x[2]^2 - 176767680*x[1]*x[3] - 201158520*x[2]*x[3] +
    63947136*x[3]^2 + 325389120*x[1]*x[4] - 77457888*x[2]*x[4] -
    216937408*x[3]*x[4] - 585806400*x[1]*x[5] - 116408124*x[2]*x[5] + 81*x[5]^2
    + 90070080*x[1]*x[6] + 16497576*x[2]*x[6] - 648*x[5]*x[6] + 3609*x[6]^2,
-103229280*x[1]*x[2] + 3456*x[2]^2 + 37386240*x[1]*x[3] + 42544872*x[2]*x[3] -
    13524760*x[3]^2 - 68819520*x[1]*x[4] + 16382304*x[2]*x[4] +
    45882080*x[3]*x[4] + 123897600*x[1]*x[5] + 24620220*x[2]*x[5] -
    19049760*x[1]*x[6] - 3489228*x[2]*x[6] + 135*x[5]*x[6] - 765*x[6]^2,
5219280*x[1]*x[2] - 216*x[2]^2 - 1890360*x[1]*x[3] - 2151168*x[2]*x[3] +
    683800*x[3]^2 + 3479520*x[1]*x[4] - 828384*x[2]*x[4] - 2319840*x[3]*x[4] -
    6264540*x[1]*x[5] - 1244862*x[2]*x[5] + 963210*x[1]*x[6] + 176418*x[2]*x[6]
    + 45*x[6]^2,
52200*x[1]*x[2] - 18900*x[1]*x[3] - 21510*x[2]*x[3] + 6840*x[3]^2 +
    34800*x[1]*x[4] - 8280*x[2]*x[4] - 23200*x[3]*x[4] - 62640*x[1]*x[5] -
    12447*x[2]*x[5] + 9630*x[1]*x[6] + 1764*x[2]*x[6]
> Genus(CG);
1
> // 3. Quotient by <W2,W27>
> G := AutomorphismGroup(X054,[W2,W27]);
> CG,prj := CurveQuotient(G);
> CG;
Curve over Rational Field defined by
0
> Ambient(CG);
Projective Space of dimension 1
Variables : $.1, $.2
```

114.8 Function Fields

An integral curve C has a coordinate ring that is an integral domain. The function field of the curve is the corresponding field of fractions in the affine case and the homogeneous degree 0 part of this in projective cases. The function field of an affine curve is isomorphic to that of its projective closure. As with schemes generally, a function field is attached to projective curves and the same object represents the function field of all of its affine patchs.

Furthermore, in the curve case, there is a unique (up to abstract scheme isomorphism) (ordinary) projective non-singular curve $\tilde{C}$ which is birationally equivalent to C (ie there are maps from C to $\tilde{C}$ which are defined at all but finitely many points and whose composite is the identity where defined) $\Leftrightarrow$ $\tilde{C}$ has the same function field (up to isomorphism) as C.

When C is projective, $\tilde{C}$ is just the *normalisation* of C. The normalisation $\tilde{C}$ differs from C only at singular points of the latter and C can be thought of as a singular model of $\tilde{C}$ and, as is usual with curves, most of the functionality provided by the function field and the objects attached - places, divisors etc. - can be more properly thought of as relating to $\tilde{C}$. The first section below treats function fields and some basic functions related to them and their elements.

From now on we assume that the reader is familiar with the notion of divisor, linear equivalence and their relationship with function fields. If not, there are very brief discussions of them at the beginning of each section and also in the introduction to this chapter, but you may also consult standard texts such as [Har77] Chapter II, 6 (especially from page 136) and Chapter IV for a more serious treatment.

For functions working with elements of a function field of a scheme and the scheme itself see Section 112.6.

114.8.1 Function Fields

For the purposes of this section, function fields are fields of rational functions on a curve C. Let f be an element and p a nonsingular rational point of C. Then one can evaluate f at p and compute the order of a zero or a pole of f at p, an integer which is positive for zeros and negative for poles. This allows points of the curve to be considered as valuations of the function field.

In fact, the proper language for discussing valuations and function fields is that of *places* and *divisors*, which really correspond to points of the projective normalisation $\tilde{C}$ of C and formal sums of these, and they are discussed in later sections. Functions in this section which take a point of a curve as an argument are convenient shorthands for functions taking a place and are only allowed when there is no ambiguity about which place is intended, which is why p is required to be non-singular. Functions which compute the zeros and poles of rational functions properly return divisors, so will be discussed later.

Finally, function fields and divisor groups are cached so that recomputation is avoided. Although it is transparent in practice, it is worth remembering that the function fields and divisor groups are always attached to the projective model of a curve, rather than to any of the affine models. Since these are all tightly related, it doesn't make any difference. The support points of divisors will be returned as points on the projective model since they can quite easily lie at infinity on any particular affine model. For a completely clean treatment, it is possible to work exclusively with the projective model, although it is certainly not necessary. Indeed, for some time the elliptic curve machinery in Magma has happily presented affine models of curves together with projective points.

```
FunctionField(C)
```

The function field of the curve C, a field isomorphic to the field of fractions of the coordinate ring of C. It can be assigned generator names using the diamond bracket notation, as in the example below. The function field will only exist when C is integral (reduced and irreducible) and this can be checked directly or by calling the function below if it is in doubt.

```
HasFunctionField(C)
```

```
Curve(F)
```

The curve used to create the function field F, or the projective closure of that curve (if it was affine). The function field is stored on projective curves so that the same field is returned whenever it is called for from any patch of the projective curve.

```
F ! r
```

Coerce the ring element r into the function field F of a scheme. For the coercion to be successful r must be in a ring related to the scheme of F, e.g. the base ring or coordinate ring of (or a field of fractions of) the scheme or one of its affine patches or a subscheme or super scheme.

```
ProjectiveFunction(f)
```

Return the function f in the function field of a scheme as a function in projective coordinates (as an element in the field of fractions of the coordinate ring of the projective scheme having function field the parent of f).

Example H114E23

After creating a curve in the usual way we make its function field F.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^4 + 2*x*y^2*z + 5*y*z^3);
> F<a,b> := FunctionField(C);
> F;
Function Field of Curve over Rational Field defined by
x^4 + 2*x*y^2*z + 5*y*z^3
> Curve(F);
Curve over Rational Field defined by
x^4 + 2*x*y^2*z + 5*y*z^3
> b^2;
```

```
b^2;
```

Once constructed, the function field will be stored with the curve (or its projective closure). Thus the same field will be returned from multiple function calls.

```
> FunctionField(C) eq FunctionField(AffinePatch(C,3));
true
```

`p @ f`

`f(p)`

`Evaluate(f, p)`

The ring element $f(p)$ where f is an element of the function field of the curve on which p is a point. If f has a pole at p the value infinity is returned.

`Expand(f, p)`

Given an element f on a curve C and a place p of C return a series which is the expansion of f at p and the uniformizing element of p.

`Completion(F, p)`

`Precision` RngIntElt *Default :* 20

The completion of the function field F of the curve C at the place p of C and a map from F into its completion.

`Degree(f)`

Given an element f of the function field of a curve, return the degree of f. This is the degree of the map given by f to $\mathbf{P}^1$ or the degree of the numerator and denominator of the principal divisor of f. If f is constant, then 0 is returned.

`Valuation(f, p)`

The degree of the zero of the function f at the point p where f is a function on the curve on which p is a point. A negative value indicates there is a pole of f at p.

`Valuation(p)`

The valuation of the function field of the curve on which p lies centred at the point p. This is a map from the function field to the integers.

`UniformizingParameter(p)`

A rational function on the curve of the nonsingular point p which having valuation 1 at p.

`Module(S)`

`Preimages`	BoolElt	*Default :* `false`
`IsBasis`	BoolElt	*Default :* `false`

Given a sequence S of elements of a function field of a curve C return the module over the base ring of C generated by the elements of S. Also return the map from the module to the function field and a sequence of preimages of the elements of S if `Preimages` is `true`.

If `IsBasis` is `true` then the elements of S will be assumed to be a basis of the module.

`Relations(S)`

`Relations(S, m)`

Given a sequence S of elements of a function field of a curve C return the module over the base ring R of C of R-linear relations between the elements of S.

`Genus(C)`

The genus of the curve C.

`FieldOfGeometricIrreducibility(C)`

Return the algebraic closure of the base ring of C in the function field of C along with the map including the closure in the function field.

`IsAbsolutelyIrreducible(C)`

Returns `true` if the field of geometric irreducibility of the curve C is the base ring of C.

`DimensionOfFieldOfGeometricIrreducibility(C)`

The dimension of the field of geometric irreducibility of the curve C over the base ring of C.

Example H114E24

Having made a curve and its function field we make an element of the function field using its named generators a, b. The function is put into a convenient form which, for this F, ensures that the denominator is a polynomial in a alone.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^4 + 2*x*y^2*z + 5*y*z^3);
> F<a,b> := FunctionField(C);
> f := a/b;
> f;
(-2*a*b - 5)/a^3
```

Now we choose a point of the curve and find that f has a pole there of order 3.

```
> p := C ! [0,0,1];
> Evaluate(f,p);
```

```
Infinity
> Valuation(f,p);
-3
```

Computing the valuations of the generators we notice that a is a uniformising parameter at p — indeed, it is the parameter automatically returned. Clearly the valuation of $f = a/b$ at p should be $1 - 4 = -3$ as computed in the previous line.

```
> vp := Valuation(p);
> vp(a), vp(b);
1 4
> UniformizingParameter(p);
a
```

`GapNumbers(C)`

The gap numbers of the curve C.

`WronskianOrders(C)`

The Wronskian orders of the curve C.

`NumberOfPlacesOfDegreeOverExactConstantField(C, m)`

`NumberOfPlacesDegECF(C, m)`

The number of places of degree m of the curve C defined over a finite field. Contrary to the `Degree` function the degree is here taken over the field of geometric irreducibility.

`NumberOfPlacesOfDegreeOneOverExactConstantField(C)`

`NumberOfPlacesOfDegreeOneECF(C)`

The number of places of degree one of the curve C defined over a finite field. Contrary to the `Degree()` function the degree is here taken over the field of geometric irreducibility.

`NumberOfPlacesOfDegreeOneOverExactConstantField(C, m)`

`NumberOfPlacesOfDegreeOneECF(C, m)`

The number of places of degree one in the constant field extension of degree m of the curve C. Contrary to the `Degree()` function the degree is here taken over the field of geometric irreducibility.

```
NumberOfPlacesOfDegreeOneECFBound(C)
NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(C)
NumberOfPlacesOfDegreeOneECFBound(C, m)
NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(C, m)
```

An upper bound on the number of places of degree one in the constant field extension of degree m (if given) of the curve C. Contrary to the `Degree` function the degree is here taken over the respective exact constant fields.

```
DivisorOfDegreeOne(C)
```

Return a divisor of the curve C of degree 1 over its field of geometric irreducibility.

```
SerreBound(C)
SerreBound(C, m)
IharaBound(C)
IharaBound(C, m)
```

The Serre and Ihara bounds of the number of places of degree 1 over the field of geometric irreducibility of the curve C over the extension of degree m of the base ring of C, which must be a finite field.

```
LPolynomial(C)
LPolynomial(C, m)
ZetaFunction(C)
ZetaFunction(C, m)
```

The L-polynomial and the ζ function of the curve C over the extension of degree m of the base ring of C, which must be a finite field.

114.8.2 Representations of the Function Field

The function field of a scheme has very little direct functionality. But when the scheme is a curve the function field is isomorphic to an algebraic function field as described in Chapter 42. This isomorphism is used internally in many computations for curves. Although the isomorphic algebraic function field can be retrieved from the function field of the scheme, this should not be necessary during ordinary usage of the curves.

```
AlgorithmicFunctionField(F)
```

Given the function field F of a curve C, this function returns the background algebraic function field, AF. As this is the object where such calculations as those involving places, divisors and differentials are performed, we refer to it as the algorithmic or arithmetic function field. Since there are curve functions which provide an interface to most of the functionality of this field via C, F and its elements, the user can usually avoid accessing AF directly. However, when it is required, it is also usually necessary to translate between elements of F and AF. A map from F to AF is thus also returned. This map is invertible and its inverse is used to map elements the other way.

```
FunctionFieldPlace(p)
CurvePlace(C, p)
FunctionFieldDivisor(d)
CurveDivisor(C, d)
FunctionFieldDifferential(d)
CurveDifferential(C, d)
```

Return the place, divisor or differential of the algebraic function field corresponding to the place, divisor or differential of a curve or convert the place p, divisor or differential d of an algebraic function field into a place, divisor or differential of the curve C.

114.8.3 Differentials

The space of differentials in Magma is the vector space of elements *df* over the function field of a curve, where f is any element of the function field and where the operator d satisfies the usual derivation conditions. This vector space is called the *differential space* and corresponds to the Kähler differentials of [Har77], II.8. Note that the differential space is not explicitly a vector space in Magma. Rather, as so often when there are many different structures to be considered, a vector space together with a map to the space of differentials is given. (Of course, basic vector space arithmetic works on the space of differentials.) In fact, this is appropriate: after all, Kähler differentials are merely a model of an object which one might prefer to define by its universal properties.

114.8.3.1 Creation of Differentials

```
DifferentialSpace(C)
```

The space of differentials of the curve C.

```
SpaceOfDifferentialsFirstKind(C)
SpaceOfHolomorphicDifferentials(C)
```

Given a curve C, this function returns a vector space V and a map from V to the space of differentials of C with image the holomorphic differentials on C.

```
BasisOfDifferentialsFirstKind(C)
BasisOfHolomorphicDifferentials(C)
```

Given a curve C, this function returns a basis for the space of holomorphic differentials of C.

```
DifferentialSpace(D)
```

Given a divisor D associated with curve C, this function returns a vector space V and a map from V to the space of differentials of the curve C containing the divisor D with image the differentials of $\omega_C(D)$. Colloquially, this refers to the differentials whose zeros are at least the positive (or effective) part of D and whose poles are no worse than the negative part of D.

```
DifferentialBasis(D)
```

Given a divisor D on a curve, this function returns the basis of the differential space of D.

```
Differential(a)
```

The exact differential $d(a)$ of the function field element a.

114.8.3.2 Operations on Differentials

The space of differentials admits vector space operations over the function field of the curve. As such, it is one-dimensional so one can even divide two non-zero differentials to recover an element of the function field.

```
Identity(S)
```

The identity differential of the differential space S of a curve.

```
Curve(S)
```

The curve for which S is the space of differentials.

```
Curve(a)
```

The curve to which the differential a belongs.

```
f * x      x * f      x + y      - x      x - y      x / r      x / y
```

The basic arithmetic in the space of differentials. Thought of as a vector space over the function field, this space is one-dimensional. The final operation uses this fact to return a function field element as the quotient of two differentials.

```
S eq T
```

Returns `true` if and only if the two spaces of differentials S and T are the same.

```
a eq b
```

Returns `true` if and only if the differentials a and b are equal.

```
a in S
```

Returns `true` if and only if a is an element of the differential space S of a curve.

```
IsExact(a)
```

Returns `true` if and only if a is known to be an exact differential, that is, if it is known to be of the form df. If this is not already known, no further attempt is made to determine that.

```
IsZero(a)
```

Returns `true` if and only if the differential a is the zero differential.

```
Valuation(d, P)
```

The valuation of the differential d of a curve at the place P of the same curve.

```
Residue(d, P)
```

The residue of the differential d of a curve at the place P of the same curve.

```
Divisor(d)
```

The divisor $(f) + (dx)$ of the differential $d = f dx$ of a curve.

```
Module(L)
```

`IsBasis`	BoolElt	*Default :* `false`
`PreImages`	BoolElt	*Default :* `false`

Given a sequence L of differentials of a curve C, return the module over the base ring of C generated by the differentials in L as an abstract module and a map from the module into the space of differentials of C.

If the parameter `IsBasis` is set to `true` then the elements in L are assumed to be a basis for the module returned. If `PreImages` is set to `true` then a sequence of the preimages of the basis elements is also returned.

```
Relations(L)
```

```
Relations(L, m)
```

Given a sequence L of differentials of a curve C, return the module over the base ring R of C of R-linear relations between the elements of L. If given, the argument m is used to compute a generating system for the relation module such that the corresponding generating system of $\{ \sum_{i=1}^{m} v_i a_i \mid v = (v_i)_i \in V \}$ consists of “small” elements where a_i are the elements of L.

```
Cartier(a)
```

```
Cartier(a, r)
```

Given a differential a belonging to a curve C and a positive integer r, this function returns the result of applying the Cartier divisor to a r times (or just once if the argument r is omitted). More precisely, let C be a curve over the perfect field k with function field F, $x \in F$ be a separating variable and $a = g\,dx \in \Omega(C)$ with $g \in F$ be a differential. The Cartier operator is defined by

$$CA(a) = \left(-d^{p-1}g/dx^{p-1}\right)^{1/p} dx.$$

This function computes the r-th iterated application of CA to a.

```
CartierRepresentation(C)
```

```
CartierRepresentation(C, r)
```

Given a curve C and a positive integer r, this function determines a row representation matrix for action of the Cartier operator on a basis of the space of holomorphic differentials of C, (applied r times). More precisely, let C be a curve over the perfect field k, $\omega_1, \ldots, \omega_g \in \Omega(C)$ be a basis for the holomorphic differentials and $r \in \mathbf{Z}^{\geq 1}$. Let $M = (\lambda_{i,j})_{i,j} \in k^{g \times g}$ be the matrix such that

$$CA^r(\omega_i) = \sum_{m=1}^{g} \lambda_{i,m}\omega_m$$

for all $1 \leq i \leq g$. This function returns M and $(\omega_1, \ldots, \omega_g)$.

Example H114E25

In this example we create a curve known to have genus 3 (a nonsingular plane quartic). So it should have a three-dimensional space of holomorphic differentials.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^4+y^4+z^4);
> Omega_C,phi := SpaceOfHolomorphicDifferentials(C);
> Omega_C;
KModule of dimension 3 over Rational Field
> F<a,b> := FunctionField(C);
> phi;
Mapping from: ModFld: Omega_C to Space of differentials of F
```

That is good. Now we make a differential and check whether it is exact (which it obviously is since that's how we made it).

```
> f := a/b;
> df := Differential(f);
> df;
(-a/(b^6 + b^2)) d(b)
> Curve(df) eq C;
true
> IsExact(df);
true
```

114.9 Divisors

We work implicitly on the resolution of a particular, usually singular, model of a curve C that has been referred to as $\tilde{C}$. To handle prime divisors at the singularities properly we use the notion of *places* of the curve and devote the first section below to their construction. Places include the case of prime divisors at points of degree greater than 1, that is, points whose coordinates do not lie in the base ring. Following this are sections on constructions of divisors and their basic arithmetic. It may seem a little strange at first to distinguish places from other divisors, but in practice when doing arithmetic the difference is not noticeable.

The most important invariant associated with a divisor D is its *Riemann–Roch* space, often denoted by $L(D)$ or $H^0(D)$. This is a vector subspace of the function field of a curve. Its computation has an enormous number of applications. One we give as an illustration is the computation of the canonical embedding of C (in the case that C is non-hyperelliptic).

This section, together with Section 114.10, is devoted to Magma's facility to work with divisors on curves. A *divisor* on a nonsingular curve C is a formal sum of points $\Sigma n_i p_i$, where each $p_i \in C$ and each n_i is an integer. It is clear that divisors form a group under componentwise addition. This group, and various variants of it, is an important invariant of the curve C. For a singular (but still irreducible) curve, one can make a similar definition in which *points* are replaced by *places*, a notion that makes precise the vague idea that singularities arise in different ways, both by "gluing" nonsingular points together and by "pinching" tangent vectors at a particular point. The sections of this chapter contain very brief discussions which will help to orient the user who already knows something about divisors. For a more complete account one should consult an advanced textbook such as [Har77] Chapter II 6, or [Ful69] Chapter 8.

We start with a description of *prime divisors* or *places*. Then we show how to create divisor groups and divisors in them and go on to explain the basic arithmetic of divisors. All of the functions here are based on equivalent functions which apply to algebraic function fields. All calculations are done in that context using the functionality of Chapter 42. A translation is made in the background for all functions described here. For information about function fields, their representations and translations see Section 114.8. The translations should not be explicitly needed in the ordinary course of working with curves and their divisors.

The main concern when working with divisors is with questions of linear equivalence. In this respect, one should at least have in mind that the most substantial calculations really make sense on projective curves. However, the functions below allow constructions using an affine curve and its points which are reinterpreted in terms of the projective closure.

Section 114.10 describes those functions which are related to linear equivalence of divisors. A divisor is called *principal* if it is linearly equivalent to the zero divisor. In the case of a curve defined over a finite field, we can compute the class group, that is, the group of divisors modulo principal divisors. In fact, we compute a finitely presented abelian group isomorphic to the class group and a map which transforms elements between the divisor group and the class group.

For any divisor on any curve, there are functions to compute Riemann-Roch spaces and a host of related entities.

114.9.1 Places

A *place* is a point of the resolution of a curve or, equivalently, a valuation of the function field of a curve. It is characterised by a point on the curve and, if that point is singular, some data that distinguishes a single resolved point above it.

114.9.1.1 Sets of Places

```
Places(C)
```

The set of places of the curve C.

```
Curve(P)
```

The curve on which the places in the set of places P lie. This will be a projective curve.

```
P eq Q
P ne Q
```

Returns `true` if and only if the sets of places P and Q are (not) equal.

114.9.1.2 Places

There are some explicit methods for constructing places. As we show later, places arise implicitly as components of divisors and this is a common way of getting hold of them.

```
Places(C, m)
```

A sequence of all places of degree m on C, a curve defined over a finite field, where m is a positive integer.

```
HasPlace(C, m)
RandomPlace(C, m)
```

If a place of degree m exists on the curve C over a finite field, then this returns `true` and a single such place, where m is a positive integer. Otherwise it returns `false`.

```
Place(p)
```

The place corresponding to the nonsingular point p of some curve.

```
Places(p)
```

A sequence containing the places corresponding to the point p of some curve.

```
Place(C, I)
```

Return the place of the curve C defined by the ideal I of the coordinate ring of C.

```
WeierstrassPlaces(C)
```

The Weierstrass places of the curve C, which are the Weierstrass places of the zero divisor of C.

`Place(Q)`

The place of a curve C determined by the sequence of elements of the sequence Q, which should all be contained in the function field of C.

`Ideal(P)`

Given a place P of a curve C return the prime ideal of the coordinate ring of the ambient of C of coordinate functions which vanish at the place P.

`TwoGenerators(P)`

Return two elements of the function field of P which determine the place P. The sequence containing these two elements can be used as input to `Place` to create a place equal to P.

Example H114E26

In this example we show how to use rational functions to create a place on a curve. This is not directly a very geometric operation. However, it is very useful since a pair of rational functions which determine a place form a very concise description of that place. Thus one often uses this method to recreate a given place on a curve in one step. We illustrate that by first finding a place and later recreating it from rational functions.

```
> P2<x,y,z> := ProjectivePlane(FiniteField(17));
> C := Curve(P2,x^5 + x^2*y^3 - z^5);
> p := C ! [1,0,1];
> Places(p);
[
    Place at (1 : 0 : 1)
]
> P := $1[1];
> P:Minimal;
Place at (1 : 0 : 1)
> TwoGenerators(P);
$.1 + 16
$.1^2*$.2
```

So now we have a place and some rational functions. As usual, these functions are elements of the function field of the curve C, so to be able to read them conveniently we assign names to that function field.

```
> FC<a,b> := FunctionField(C);
> TwoGenerators(P);
a + 16
a^2*b
```

We can use this sequence to recreate the place P. The real convenience of the first line of code below is that it could be in a different MAGMA session in which only the curve C has been defined together with the names a, b of its function field. (You can confirm this by running the four relevant lines in a separate Magma session.) The final line is simply to confirm that we really have created the same place P as we started with.

```
> Place([a+16,a^2*b]);
Place at (1 : 0 : 1)
> Place([a+16,a^2*b]) eq P;
true
> Place([a+16,a*b,a^2*b^2]) eq P;
true
```

Notice that in the final line we create exactly the same place using more than the two elements that `TwoGenerators` returned.

`Zeros(f)`

`Poles(f)`

A sequence of places of the curve C containing the zeros or poles of f where f is an element of the function field of C.

`Zeros(C, f)`

`Poles(C, f)`

A sequence of places of the curve C containing the zeros or poles of f where f is some function on C, i.e. f is coercible into the function field of C.

`CommonZeros(L)`

`CommonZeros(C, L)`

Given a sequence L of elements of the function field of some curve C or a curve C and a sequence L of functions on C, return the zeros which are common to all elements of L as places of C.

Example H114E27

The second argument to the intrinsic `Poles(C,f)` can be either an element of the function field of the curve C or an element of the function field of its ambient space.

```
> A<x,y> := AffineSpace(GF(2),2);
> C := Curve(A,x^8*y^3 + x^3*y^2 + y + 1);
> FA<X,Y> := FunctionField(A);
> FC<a,b> := FunctionField(C);
> Poles(C,X/Y);
[
    Place at (1 : 0 : 0)
]
> Poles(C,a/b);
[
    Place at (1 : 0 : 0)
]
> $1 eq $2;
```

```
true
```

In particular, we did not use the ambient coordinates x, y in the arguments.

`p1 + p2` `- p1` `p1 - p2` `k * p`

`p div k` `p mod k` `Quotrem(p1, k)`

`Curve(P)`

The projective curve on which the place P lies.

`RepresentativePoint(P)`

A representative point of the projective model of the curve underlying the place.

`P eq Q`

`P ne Q`

Returns `true` if and only if the two places are (not) the same.

`P in S` `P notin S`

`Valuation(f, P)`

The order of vanishing of the function f on the curve C at the place P of C. A negative value indicates a pole at P.

`Valuation(P)`

The valuation of the function field centred at the place P. This is a map from the function field to the integers.

`Valuation(a, P)`

The valuation of the differential a at the place P.

`Residue(a, P)`

The residue of the differential a at the degree 1 place P.

`UniformizingParameter(P)`

A function on the curve of the place P having valuation 1 at P.

`IsWeierstrassPlace(P)`

`IsWeierstrassPlace(D, P)`

Returns `true` if and only if the place P (which must have degree 1) is a Weierstrass place (of divisor D if given).

`ResidueClassField(P)`

The residue class field of the place P.

`Evaluate(a, P)`

Evaluate the element a in the function field of the curve of the place P, returning an element of the residue class field of P.

`Lift(a, P)`

Lift the element a of the residue class field of the place P (including infinity) to a function on the curve of P.

`Degree(P)`

The degree of the place P of a curve over the base ring of the curve of P.

`GapNumbers(C, P)`

`GapNumbers(P)`

The gap numbers of the curve C at the degree 1 place P.

`Parametrization(C, p)`

`Parametrization(C, p, P)`

Returns a map parametrizing the rational curve C at the rational point p or place p of degree 1. If p is a *singular* point on C, then it must have a unique place above it of degree 1. If P is also given it must be the projective line of dimension 1 as a curve (ie of type `Crv`) and then the domain of the map will be P.

114.9.2 Divisor Group

A curve has an associated group of divisors which is simply the formal abelian group generated by the places of the curve C. Divisors are elements of this group. In other words, divisors are expressions of the form $\Sigma n_i p_i$ where n_i are integers, p_i are places of the curve which one usually assumes to be distinct. Each term np is called a *summand* of d or the *component of* d *corresponding to* p. The integer n will be called the *coefficient* or *multiplicity* of the summand.

Divisors are created by specifying the curve for which they will a divisor (if that is not clear) and then giving sufficient data to identify precisely the divisor in question. This data could be a list of points or places together with integers, but there are many other creation methods. Divisors are printed as a linear combination of the places which support them, if such a combination is known. However, giving this information can be extremely expensive so often printing simply refers to the curve.

`DivisorGroup(C)`

The group of divisors of the curve C. This curve may be either an affine or projective curve.

`Curve(Div)`

The curve that was used to create the divisor group Div, or its projective model.

`Div1 eq Div2`

`Div1 ne Div2`

Returns `true` if and only the divisor groups *Div*1 and *Div*2 are (not) equal.

114.9.3 Creation of Divisors

`DivisorGroup(D)`

The divisor group in which the divisor D lies.

`Curve(D)`

The (projective) curve on which the divisor D lies.

`Identity(D)`

`Id(D)`

`D ! 0`

The zero divisor of the divisor group D of a curve.

`Div ! p`

`Divisor(p)`

The prime divisor in the divisor group *Div* of the curve C corresponding to the place or nonsingular point p of some curve C.

`Divisor(D, S)`

`Divisor(C, S)`

`Divisor(S)`

The divisor of the curve C or the curve of the divisor group D described by the factorization sequence S. The sequence should contain tuples of the form `<place,integer>`.

Example H114E28

One can use the intrinsic `Divisor(S)` to reconstruct a divisor from concise data related to it. The intrinsics `Support` and `CanonicalDivisor` are defined below.

```
> P<x,y,z> := ProjectivePlane(FiniteField(17));
> C := Curve(P,x^5 + x^2*y^3 - z^5);
> F<a,b> := FunctionField(C);
> K := CanonicalDivisor(C);
> supp, exps := Support(K);
> Q := [ < RationalFunctions(supp[i]),exps[i] > : i in [1..#supp] ];
> Q;
[
    <[ a, 2*a^2*b^2 + 4*a*b ], 2>,
    <[ a + 16, a^2*b ], 2>,
    <[ a^4 + a^3 + a^2 + a + 1, a^2*b ], 2>,
    <[ 1/a, (a + b)/a ], -2>,
    <[ 1/a, (a^2 + 16*a*b + b^2)/a^2 ], -2>
]
> K;
Divisor 2*Place at (0 : 1 : 0) + 2*Place at (1 : 0 : 1) + 2*Place at
($.1 : 0 : 1) - 2*Place at (16 : 1 : 0) - 2*Place at (16*$.1 + 1 : 1 : 0)
```

Now we can reconstruct the divisor K using this sequence.

```
> Divisor([<Place(f[1]), f[2]> : f in Q]);
Divisor on Curve over GF(17) defined by
x^5 + x^2*y^3 + 16*z^5
> K eq $1;
true
```

```
PrincipalDivisor(C, f)
PrincipalDivisor(D, f)
PrincipalDivisor(f)
Divisor(C, f)
Divisor(D, f)
Divisor(f)
```

The principal divisor corresponding to f, that is, the divisor of the curve C of zeros and poles of the function field element f, where C is the curve of the divisor group D if given.

```
Divisor(a)
```

The divisor of the curve C corresponding to the differential a of C.

```
Divisor(C, X)
Divisor(D, X)
```

The divisor described by intersection of the curve C with the scheme X, (where C is the curve of the divisor group D if the group is given instead of the curve).

```
Divisor(C, p, q)
Divisor(D, p, q)
```

The principal divisor corresponding to the line through points p and q (the tangent line to the curve C there if they coincide) where C is the curve of the divisor group D if given.

```
Divisor(C, I)
Divisor(D, I)
```

The divisor of the curve C defined by the ideal I of the ambient coordinate ring where C is the curve of the divisor group D if given.

```
Decomposition(D)
```

The decomposition sequence of D as a sequence of tuples of the form `<place, multiplicity>` characterizing the divisor D.

```
Support(D)
```

The sequence of places in the support of D, followed by their sequence of multiplicities in D.

Example H114E29

A curve, its divisor group and some divisors are created.

```
> P<x,y,z> := ProjectiveSpace(GF(7), 2);
> C := Curve(P,y^2*z - x^3 - x*z^2 - z^3);
> Div := DivisorGroup(C);
> Div;
Group of divisors of Curve over GF(7) defined by
6*x^3 + 6*x*z^2 + y^2*z + 6*z^3
> FP<a,b> := FunctionField(P);
> D := Divisor(C,a);
> D;
Divisor of Curve over GF(7) defined by
6*x^3 + 6*x*z^2 + y^2*z + 6*z^3
> Decomposition(D);
[
    <Place at (0 : 1 : 0), -2>,
    <Place at (0 : 6 : 1), 1>,
    <Place at (0 : 1 : 1), 1>
]
> D;
Divisor -2*Place at (0 : 1 : 0) + 1*Place at (0 : 6 : 1) + 1*Place at (0 : 1 : 1)
```

The support of a divisor is written in the style of the factorization of other objects in Magma; compare with the factorization of the integer 84 below. This expression is called the *factorization* of a divisor and provides a method of accessing the individual components.

```
> Factorization(84);
[ <2, 2>, <3, 1>, <7, 1> ]
> Support(D)[2];
Place at (0 : 6 : 1)
```

One can access the point underlying a given place.

```
> p := Support(D)[1];
> p;
Place at (0 : 1 : 0)
> RepresentativePoint(p);
(0 : 1 : 0)
```

`CanonicalDivisor(C)`

A divisor in the canonical divisor class of the curve C.

`RamificationDivisor(C)`

The ramification divisor of the curve C.

114.9.4 Arithmetic of Divisors

`D + E` `- D` `D - E` `n * D` `D div n` `D mod n`

Basic formal arithmetic of divisors; D and E are divisors (or places) and n is an integer.

`Quotrem(D, n)`

The quotient and remainder on dividing the divisor D by the integer n.

`Degree(D)`

The sum of coefficients of the divisor D multiplied by the degrees of the places of the corresponding components.

`IsEffective(D)`

`IsPositive(D)`

Returns `true` if and only if all coefficients of the divisor D are nonnegative.

`Numerator(D)`

`Denominator(D)`

The numerator, respectively denominator, of the divisor D of a curve. The numerator and denominator are both positive divisors such that D is the difference between numerator and denominator.

`SignDecomposition(D)`

The minimal effective divisors A and B such that the equality of divisors $D = A - B$ holds.

Example H114E30

The sign decomposition of the previous example is calculated.

```
> P<x,y,z> := ProjectiveSpace(GF(7),2);
> C := Curve(P,y^2*z - x^3 - x*z^2 - z^3);
> Div := DivisorGroup(C);
> phi := hom< Parent(x/z) -> FP | [FP.1,FP.2,1] >
>             where FP is FunctionField(P);
> D := Divisor(Div,phi(x/z));
> Decomposition(D);
[
    <Place at (0 : 1 : 0), -2>,
    <Place at (0 : 6 : 1), 1>,
    <Place at (0 : 1 : 1), 1>
]
> Decomposition(D div 2);
[
    <Place at (0 : 1 : 0), -1>
]
> A, B := SignDecomposition(D);
> IsEffective(A);
true
> IsEffective(B);
true
> A - B eq D;
true
```

`d in D`   `d notin D`

`D eq E`

`D ne E`

Returns `true` if and only the divisors D and E are (not) equal. Note that this means equality in the group of divisors and is not the same as being linearly equivalent.

`D lt E`   `D le E`   `D gt E`   `D ge E`

`IsZero(D)`

Returns `true` if and only if all coefficients of the divisor D are zero.

`IsCanonical(D)`

Returns `true` if and only if the divisor D is the divisor of a differential, in which case also return a differential realising this.

`GCD(D1, D2)`

`Gcd(D1, D2)`

`GreatestCommonDivisor(D1, D2)`

The greatest common divisor of the divisors $D1$ and $D2$. This is the divisor supported on the places common to the support of both divisors with coefficients the minimum of those occurring in $D1$ and $D2$.

`LCM(D1, D2)`

`Lcm(D1, D2)`

`LeastCommonMultiple(D1, D2)`

The least common multiple of the divisors $D1$ and $D2$. This is the divisor supported on all the places in the supports of $D1$ and $D2$ with coefficients the maximum of those occurring in the input divisors.

Example H114E31

We find that a given divisor is actually a canonical divisor.

```
> P<x,y,z> := ProjectiveSpace(GF(7),2);
> C := Curve(P,y^2*z - x^3 - x*z^2 - z^3);
> Div := DivisorGroup(C);
> phi := hom< Parent(x/z) -> FP | [FP.1,FP.2,1] >
>                where FP is FunctionField(P);
> D := Divisor(Div,phi(x/z));
> IsCanonical(D);
true (($.1) ^ 1 * ($.1^3 + $.1 + 1) ^ -1 * ($.1) ^ 1) d($.1)
```

The printing of the differential in the last line above is not very clear since names have not been assigned, but nonetheless, it can be used as an argument to intrinsics.

```
> _, dd := IsCanonical(D);
> Valuation(dd,Support(D)[1]);
-2
```

114.9.5 Other Operations on Divisors

`Ideal(D)`

Given a divisor D of a curve C, return the ideal of the coordinate ring of the ambient of C of coordinate functions which cuts out D.

`Valuation(D,p)`

`Valuation(D,P)`

The coefficient of the divisor summand of the divisor D corresponding to the point p or place P.

```
ComplementaryDivisor(D,p)
```

```
ComplementaryDivisor(D,P)
```

The divisor after removing from the divisor D the component corresponding to the point p or place P.

114.10 Linear Equivalence of Divisors

114.10.1 Linear Equivalence and Class Group

```
IsPrincipal(D)
```

Returns `true` if and only if the divisor D is the divisor of zeros and poles of some rational function. A rational function which performs that role will also be returned. Recall that any two such functions differ only by a scalar factor.

```
IsLinearlyEquivalent(D1,D2)
```

Returns `true` if and only if the difference $D1 - D2$ of the two divisor arguments is a principal divisor. In that case return also the rational function giving this equivalence.

```
IsHypersurfaceDivisor(D)
```

For an effective hypersurface divisor D on an ordinary projective curve C, returns `true` if and only if D is the scheme theoretic intersection of C with a hypersurface H of the ambient projective space. If so, also returns an equation for H and the degree of H.

Example H114E32

We check that an effective canonical divisor on a non-singular degree 4 plane curve is indeed a divisor coming from the intersection of the curve with a hyperplane.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^3*y+y^3*z+z^3*x);
> D0 := CanonicalDivisor(C);
> f := Basis(D0)[1];
> D := D0+PrincipalDivisor(f);
> IsEffective(D);
true
> IsHypersurfaceDivisor(D);
true -2/9*x
1
```

```
ClassGroup(C)
```

The *divisor class group*, or simply *class group* of a curve is the group of divisors modulo principal divisors. If C is a curve defined over a finite field, then this function returns an abelian group isomorphic to its divisor class group, a map of representatives from the class group to the divisor group and the homomorphism from the divisor group onto the class group. The second map has an inverse, so translation between the group and the divisors can be achieved using only this map.

```
ClassNumber(C)
```

The order of the class group of the curve C.

```
GlobalUnitGroup(C)
```

The group of global units of the function field of the curve C, that is the multiplicative group of the field of geometric irreducibility of C as an abelian group together with the map into the function field of C.

Example H114E33

We compute the class group of a curve defined over a finite field. The calculation takes a few seconds.

```
> A<x,y> := AffineSpace(GF(2,5),2);
> C := Curve(A,x^7 + x^4*y^3 + x*y^2 + y);
> Genus(C);
3
> Cl, _, phi := ClassGroup(C);
> Cl;
Abelian Group isomorphic to Z/26425 + Z
Defined on 2 generators
Relations:
    26425*Cl.1 = 0
```

We can use the map ϕ to pull elements of the abelian group back to the divisor group. For curves over finite fields, this is one way of constructing interesting divisors.

```
> Div := DivisorGroup(C);
> Div eq Domain(phi);
true
> D := Cl.1 @@ phi;
> D;
Divisor of Curve over GF(2^5) defined by
x^7 + x^4*y^3 + x*y^2 + y
```

The unpleasant printing with the dollar signs is happening because no names have yet been assigned to the projective closure of C. Notice that the printing is rather uninformative. This is because a factorization of d is not yet known and could be an extremely expensive computation.

```
> Decomposition(D);
[
    <Place at (0 : 0 : 1), -3>,
    <Place at ($.1 : $.1^10*$.1^2 + $.1^26*$.1 + $.1^22 : 1), 1>
]
> Degree(D);
0
```

The degree function simply returns the sum of the integer valuations in the factorization. Those valuations can be seen as the second entry of each tuple in the support.

```
ClassGroupAbelianInvariants(C)
```

The abelian invariants of the curve C.

```
ClassGroupPRank(C)
```

The p-rank of the class group of C.

```
HasseWittInvariant(C)
```

The Hasse–Witt invariant of the curve C.

114.10.2 Riemann–Roch Spaces

If D is a divisor on C then colloquially speaking the Riemann–Roch space $L(D)$ is the finite dimensional vector subspace of the function field of C consisting of functions with poles no worse that D (and at least as many zeros as the negative part of D if D is not effective). To be precise, we say that

$$L(D) = \{f \in k(C) \mid \mathrm{div}(f) + D \geq 0\}$$

where $\mathrm{div}(f)$ is the principal divisor of zeros and poles of f and the condition ≥ 0 is simply shorthand to mean that the left-hand side is an effective divisor.

For any divisor on a projective curve defined over a field this vector space is finite dimensional. Its dimension $\ell(D)$ appears in the Riemann–Roch formula

$$\ell(D) - \ell(K_C - D) = \deg(D) + 1 - g$$

where K_C is the canonical divisor and g is the genus of C.

The space of *effective* divisors linearly equivalent to D, the complete linear system of D, is the projectivisation of the Riemann–Roch space $L(D)$. This can be used to create a map from the curve C to a projective space of dimension $\ell(D) - 1$.

```
Reduction(D)
```

```
Reduction(D, A)
```

Let D be a divisor. Denote the result of both functions by $\tilde{D}$, r, A and a (for the second function the input A always equals the output A). A has (must have) positive degree and the following holds:

(i) $D = \tilde{D} + rA - (a)$,

(ii) $\tilde{D} \geq 0$ and $\deg(\tilde{D}) < g + \deg(A)$ (over the field of geometric irreducibility),

(iii) $\tilde{D}$ has minimal degree among all such divisors satisfying (i), (ii).

```
RiemannRochSpace(D)
```

A vector space V and an isomorphism from V to the Riemann–Roch space of D in the function field of the curve on which the divisor D lies.

```
Basis(D)
```

A sequence containing a basis of the Riemann-Roch space $L(D)$ of the divisor D.

```
ShortBasis(D)
```

A sequence containing a basis of the Riemann-Roch space $L(D)$ of the divisor D in short form.

```
Dimension(D)
```

The dimension $\ell(d)$ of the Riemann-Roch space of the divisor D.

```
DifferentialSpace(D)
```

Returns a vector space V and a map from V to the space of differentials of the curve C containing the divisor D with image the differentials of $\omega_C(D)$. Colloquially, this means the differentials whose zeros are at least the positive (or effective) part of D and whose poles are no worse than the negative part of D.

```
DifferentialBasis(D)
```

A basis of the space of differentials of the divisor D.

```
IndexOfSpeciality(D)
```

The index of speciality of the divisor D, that is the dimension $\ell(K_C - D)$ appearing in the Riemann–Roch formula.

```
IsSpecial(D)
```

Returns `true` if and only if the divisor D is special.

```
GapNumbers(D)
GapNumbers(D,p)
```

The gap numbers of the divisor D, at the place p if included.

```
GapNumbers(p)
```

The gap numbers of the nonsingular point p.

```
WeierstrassPlaces(D)
WeierstrassPoints(D)
```

A sequence containing the Weierstrass places (or underlying points) of the divisor D.

```
WronskianOrders(D)
```

The Wronskian orders of the divisor D.

`RamificationDivisor(D)`

The ramification divisor of the divisor D.

`DivisorMap(D)`

`DivisorMap(D,P)`

A map from the curve of the divisor D to the projective space P (which will be created in the background if not given as an argument). The dimension of P must be $\ell(D) - 1$.

`CanonicalMap(C)`

`CanonicalMap(C,P)`

The canonical map from the curve C to the projective space P (which will be created in the background if not given as an argument). This is the map determined by (a basis of the) Riemann–Roch space of the canonical divisor K_C. In particular, the dimension of P must be $\ell(K_C) - 1 = g - 1$.

`CanonicalImage(C, phi)`

`CanonicalImage(C, eqns)`

These functions compute the canonical image of a projective curve C of genus at least 2 much more efficiently than the generic image machinery applied to the canonical map. In the first function, the second argument should be the canonical map and in the alternative it should be a sequence of polynomials defining the canonical map.

A second return value is a boolean which is true if and only if the image is a rational curve or equivalently if and only if C is a hyperelliptic curve. Note that these functions may also be used for computing the rational normal curve image of a genus 0 curve C under the map given by any (non-trivial) complete linear system.

Example H114E34

We first make a curve and compute its genus.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2,X^7 + X^3*Y^2*Z^2 + Z^7);
> Genus(C);
3
```

If C is not hyperelliptic then its canonical map will embed it as a nonsingular quartic curve in the projective plane. We make the canonical map. We even include the plane $P2$ that we have already created as an argument so that it will be set as the codomain of the map.

```
> phi := CanonicalMap(C,P2);
> phi;
Mapping from: Prj: P2 to Prj: P2
with equations :
X^2 X*Z Z^2
> phi(C);
```

```
Curve over Rational Field defined by -X*Z + Y^2
```

That curve is certainly not a plane quartic. Indeed, it is evidently a rational curve, so C must have been hyperelliptic.

```
> D := phi(C);
> Genus(D);
0
```

The bicanonical map will embed C since its genus is strictly bigger than 2.

```
> D := 2 * CanonicalDivisor(C);
> phi2 := DivisorMap(D);
> Dimension(Codomain(phi2));
5
> P5<a,b,c,d,e,f> := Codomain(phi2);
> phi2(C);
Scheme over Rational Field defined by
a^2 + b^2 + e*f
-b*d + c^2
-b*e + c*d
-b*f + d^2
-b*f + c*e
-c*f + d*e
-d*f + e^2
> Dimension(phi2(C));
1
> IsNonsingular(phi2(C));
true
```

If you are familiar with the equations of rational normal curves, you will recognise this as a quadric section of a standard scroll—the cone on the rational normal curve of degree 4—which misses the vertex of the cone. This is exactly what gives the curve its hyperelliptic structure. One could go on to make a ruled surface with a map to $\mathbf{P}^5$ with this scroll as its image and pull back the curve to the ruled surface on which the ruling cuts out the degree 2 linear system of the hyperelliptic curve. This is carried out for a particular trigonal curve later in Section 114.11.1.

114.10.3 Index Calculus

Magma contains functionality to compute discrete logarithms in the degree 0 divisor class group of plane curves over finite fields. The algorithm used for this is Diem's version of index calculus [Die06], which looks for relations by considering lines through the points of the factor base. In order to use this algorithm, the group order must be know and given as input.

As is always the case with index calculus, the algorithm consists of two main stages: the sieving stage, during which relations are obtained and stored in a matrix, and the linear algebra stage, during which a non-trivial element of the kernel of this matrix is computed. It is possible to only do the sieving and compute the matrix. For this it is not necessary to know the group order. This can be used to obtain information on how fast sieving can be done. Generally, the linear algebra stage will be the bottle neck for larger examples.

The running time of the algorithm mainly depends on the degree of the curve. In [Die06] it is explained how one can obtain a model of a given curve with relatively low degree. This can greatly speed up the running time of the algorithm.

The algorithm will not work if the field is too small, as there will not be enough lines to produce relations in that case. One requires that at least $q \geq d!$ for a degree d curve over $GF(q)$.

Elements in the degree 0 class group can be represented by divisors of degree 0. But we also allow divisors of non-zero degree D as input, together with a fixed divisor $D0$ of degree 1. This will be interpreted as representing the divisor class $D - \deg(D)D0$. Note that once $D0$ is fixed, and D is effective of minimal degree, then D is uniquely determined by the divisor class. Starting out with an arbitrary divisor E, this unique D can be computed with the `Reduction(E,D0)` command.

```
IndexCalculus(D1, D2, D0, np)
```

```
IndexCalculus(D1, D2, D0, np, n, rr)
```

Compute the discrete logarithm of $D2 - \deg(D2)D0$ with base $D1 - \deg(D1)D0$. The group order np must be given. Optionally, one can also give n as approximate size of the factor base to be used, and rr as the number of required relations.

```
IndexCalculusMatrix(D1, D2, D0, n, rr)
```

Find the sparse matrix with relations found in the sieving stage. Input is the same as for `IndexCalculus`, except that the group order is not given. Outputs M, pos, fb, Da, Db, a, b, where M is the sparse relation matrix. Da and Db are divisors that split over the factor base, and $Da - \deg(Da)D0$ and $Db - \deg(Db)D0$ are linearly equivalent to $a(D1 - \deg(D1)D0)$ and $b(D2 - \deg(D2)D0)$, respectively. fb is a sequence of points that contains the support of Da, Db and $D0$. pos is a sequence of integers that indicates the position of the points of fb in the factor base. The last two rows of M correspond to the divisors $Da - \deg(Da)D0$ and $Db - \deg(Db)D0$.

```
MultiplyDivisor(n, D , D0)
```

Effective divisor E of minimal degree such that $E - \deg(E)D0$ is equivalent to $n(D - \deg(\mathrm{D})D0)$.

Example H114E35

In this example we compute a discrete logarithm in the class group of a curve of degree 6. The curve is over $GF(2^{13})$, but it is a base change of a curve over $GF(2)$

```
> A2<x,y>:=AffinePlane(GF(2));
> C1:=ProjectiveClosure(Curve(A2,x^5*y + x*y^2 + y^6 + y + 1));
> L:=LPolynomial(C1);
> Evaluate(L,1);
```

```
752
> K<z>:=CyclotomicField(13);
> np:=Numerator(Norm(Evaluate(L,z)));
> Factorisation(np);
[ <1753104484044610457180695483606558837, 1> ]
```

So the class group of $C1$ has order 752 over $GF(2)$, and is has order $752np$ over $GF(2^{13})$, where np is a large prime. Now we create some divisors, and make sure that $D2 - \deg(D2)D0$ is in the subgroup generated by $D1 - \deg(D1)D0$.

```
> F13<u>:=GF(2^13);
> C13:=ChangeRing(C1,F13);
> D1:=Divisor(C13![u^4758,u^3]);
> D2:=752*Divisor(C13![u^1325,u^6]);
> D0:=Divisor(C13![u^2456,u^11]);
```

In order to get an idea what is going on, we set the verbose flag, and we perform the discrete logarithm computation. In the end, we verify the result.

```
> SetVerbose("CurveIndexcal",1);
> time dl:=IndexCalculus(D1,D2,D0,752*np);
Try to find factor base of size 5087
Try to find 5138 relations.
First input divisor already splits.
Second input divisor already splits.
First index calculus input: Divisor 1*Place at (u^4758 : u^3 : 1)
Multiple of original input: 1
Second index calculus input: Divisor 752*Place at (u^1325 : u^6 : 1)
Multiple of original input: 1
Sieving
Number of relations found: 5138
Number of elements in factor base: 5089
Find element of kernel of matrix
Found kernel element
DLP mod 1753104484044610457180695483606558837:
12522086041061799120645274502697942
Time: 164.900
> MultiplyDivisor(dl,D1,D0) eq Reduction(D2,D0);
true
```

114.11 Advanced Examples

These examples are intended to demonstrate basic programming in Magma using the functions of this chapter together with a few from Chapter 112. There is little or no explanation of the geometry behind the examples. We assume here that you are familiar with that and are really interested in the problem of realising it in Magma.

114.11.1 Trigonal Curves

We discuss a particular example covered by Petri's theorem. In fact, we write down a curve C of genus 8 which is trigonal. We discover this easily since its canonical embedding is not cut out by quadrics. It would be nice to have automatic functions to recognise the equations of the surface scroll cut out by the quadrics, but at the moment they don't exist so we have to make that calculation by hand.

Example H114E36

First we make the curve and compute its canonical model.

```
> k := Rationals();
> P<X,Y,Z> := ProjectiveSpace(k,2);
> C := Curve(P,X^8 + X^4*Y^3*Z + Z^8);
> Genus(C);
8
> phi := CanonicalMap(C);
> P7<a,b,c,d,e,f,g,h> := Codomain(phi);
> CC := phi(C);
> CC;
a^3*e + d^4 + d^2*h^2
a^3*f + d^3*e + d*e*h^2
a^3*g + d^3*f + d*f*h^2
a^3*h + d^3*g + d*g*h^2
a^2*b + d^3 + d*h^2
a^2*c + d^2*e + e*h^2
a*b*c + d^2*f + f*h^2
a*c^2 + d^2*g + g*h^2
b*c^2 + d^2*h + h^3
-a*c + b^2  -a*e + b*d  -a*f + c*d  -a*f + b*e
-a*g + c*e  -d*f + e^2  -a*g + b*f  -a*h + c*f
-d*g + e*f  -d*h + f^2  -a*h + b*g  -b*h + c*g
-d*h + e*g  -e*h + f*g  -f*h + g^2
```

In this example, we can see all the quadrics which cut out the canonical model CC. But even if we could not, or if computing the full canonical ideal was too difficult, we can compute the conics in the canonical ideal separately using only linear algebra.

```
> SC := Image(phi,C,2);
> SC;
a*c - b^2  a*e - b*d  a*f - c*d  a*g - c*e
```

```
a*h - c*f  b*e - c*d  b*f - c*e  b*g - c*f
b*h - c*g  d*f - e^2  d*g - e*f  d*h - f^2
e*g - f^2  e*h - f*g  f*h - g^2
> Dimension(SC);
2
```

We would like to identify the scroll SC. Even better, we would like to find a map from a ruled surface to this scroll and pull the image curve CC back to this ruled surface. Then the fibres of the ruling will cut out the g_3^1 on C giving its trigonal structure. We will also see the Maroni invariant of C directly. In this case one immediately recognises the equations of the scroll so can write down the ruled surface and choose the bidegree of linear system which gives the map to the scroll.

```
> F<r,s,u,v> := RuledSurface(k,2,4);
> psi := map< F -> P7 | [ m : m in MonomialsOfWeightedDegree(F,[0,1]) ] >;
> SF := psi(F);
> DefiningIdeal(SF) eq DefiningIdeal(SC);
true
```

To realise the curve's trigonal structure, we need to create a divisor by intersecting it with a fibre of the ruling. The natural way would be to pull the curve back to F via psi and work there. However, Magma currently cannot create divisors that lie on curves on scrolls.
Luckily, we *can* work with the image `CC` in `P7` and obtain the divisor `D` of the image of a fibre under `psi` intersected with `CC` for our g_3^1. This is then used to define a $3-1$ map to the projective line.

```
> fib := psi(Scheme(F,r));
> Dimension(fib);
1
> D := Divisor(CC,fib);
> Degree(D);
3
> #Basis(D);
2
```

So `D` really does give us a g_3^1. To get the map to P^1, we can pull it back to `C`, but it is faster to compose the divisor map on `CC` with `phi`. Proceeding this way, it is then a good idea to use the function field of `C` to simplify the map description.

```
> phiD := DivisorMap(D);
> mpD := Expand(Restriction(phi,C,CC)*phiD);
> FC := FunctionField(C);
> rat := FC!(p1/p2) where p1,p2 := Explode(DefiningPolynomials(mpD));
> mpD := map<C->Codomain(mpD)|[rat,1]>; mpD;
Mapping from: CrvPln: C to Projective Space of dimension 1
Variables : $.1, $.2
with equations :
X^7 + X^3*Y^3*Z
Z^7
```

114.11.2 Algebraic Geometric Codes

Magma includes functions for working with codes which arise from algebraic geometry. Discussion of these functions is left to the chapter on error-correcting linear codes, Chapter 152. As is well known, these codes are often created using Riemann–Roch spaces of divisors on curves. Here we demonstrate the creation of such a code taken from the book of van Lint and van der Geer [vLvdG88]. This is a famous example which arises many times in that book, as Example II (3.12) for instance.

Example H114E37

This code is based on the Klein quartic over the finite field $F8$ of 8 elements, a curve that we define immediately. Notice that we start by defining a curve C over the field of 2 elements. This is so that we can investigate C over small fields while still being able to work over $F8$ later.

```
> F2 := FiniteField(2);
> F4<t4> := FiniteField(4);
> F8<t8> := FiniteField(8);
> P<x,y,z> := ProjectiveSpace(F2,2);
> C := Curve(P,x^3*y + y^3*z + z^3*x);
> C8<a,b,c> := BaseChange(C,F8);
> C8;
Curve over GF(2^3) defined by
a^3*b + a*c^3 + b^3*c
```

The code will have length 24, corresponding to the 24 rational points of C.

```
> #RationalPoints(C8);
24
```

In constructing such codes, one must have a collection of points, in this case the 24 rational points we have just found, and a divisor whose support is disjoint from these points. As the divisor, we take some multiple of a conjugate pair of points defined over the finite field of 4 elements. In Magma, it is convenient to use the function field machinery to describe this as a place of degree 2. It is constructed as the intersection of C with one of its bitangents.

```
> L := Curve(P,x+y+z);
> IntersectionPoints(C,L);
{@ @}
> C4 := BaseChange(C,FiniteField(4));
> P4 := Ambient(C4);
> L4 := BaseChange(L,P4);
> IntersectionPoints(C4,L4);
{@ (t4 : t4^2 : 1), (t4^2 : t4 : 1) @}
> [ IntersectionNumber(C4,L4,p) : p in $1 ];
[ 2, 2 ]
```

So we see that L is indeed a bitangent of C and that the two points of intersection are defined properly over the finite field of 4 elements. In order to be able to compute the Riemann-Roch space of a divisor, we compute this intersection as a place on $C8$. Notice how the support of the divisor D is not used in its display until it has been explicitly computed.

```
> FP8<X,Y> := FunctionField(Ambient(C8));
> Div := DivisorGroup(C8);
> D := Divisor(Div,X + Y + 1);
> D;
Divisor on Curve over GF(2^3) defined by
a^3*b + a*c^3 + b^3*c
> Decomposition(D);
[
    <Place at (0 : 1 : 0), -3>,
    <Place at (1 : 0 : 0), -1>,
    <Place at ($.1 + 1 : $.1 : 1), 2>
]
> D;
Divisor -3*Place at (0 : 1 : 0) + 2*Place at ($.1 + 1 : $.1 : 1) -
1*Place at (1 : 0 : 0)
```

This divisor contains the place we require with multiplicity 2 and some negative contributions at infinity. We extract the degree 2 place we need.

```
> p := Support(D)[2];
> Degree(p);
2
```

Now we can make the code. Rather than using the rational points of $C8$, the code creation intrinsic takes a sequence of degree 1 places. But they are also easily retrieved. The divisor we use will be $3p$.

```
> G := 3*p;
> S := Places(C8,1);
> #S;
24
> AGC := AlgebraicGeometricCode(S,G);
> Dimension(AGC);
4
```

In fact, it is the dual code that is really wanted, but this is easy.

```
> Dual(AGC);
[24, 20] Linear Code over GF(2^3)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^5 t8 t8^5 t8^3]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^5 t8^2 t8^6 t8^5]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^3 t8^6 t8^5]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 t8^6 t8^3 t8^4]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^2 t8^3 t8 t8^2]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^6 t8^6 t8^2 t8^6]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^2 t8^4 t8^4 t8^6]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 t8^6 t8^3 t8^3 t8^2]
```

```
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 t8^4 0 t8^4]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 t8^4 t8^3 t8^6 1]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 t8 t8^2 t8^2 t8^3]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 t8^6 t8^6 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 t8^2 t8^4 t8^3 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 t8^6 t8 t8^2 t8]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 t8^5 t8 t8^6]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 t8 1 1 t8^3]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 t8^3 1 t8^5 t8^2]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 t8^2 t8^2 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 t8^3 t8 t8^5 t8^5]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 t8^4 t8^3 t8 t8^4]
```

114.12 Curves over Global Fields

114.12.1 Finding Rational Points

There are also various point search routines for more general schemes.

PointsCubicModel(C, B : *parameters*)

OnlyOne	BoolElt	*Default :* false
ExactBound	BoolElt	*Default :* false
Verbose	CubicModelSearch	*Maximum :* 1

Given a plane cubic over the rationals, this function searches, by a reasonably efficient method due to Elkies [Elk00], for a point on C of naïve height up to B — the asymptotic running time is $O(B)$.

If OnlyOne is set to true, the search stops as soon as it finds one point; however, the algorithm is p-adic and there is no guarantee that points with small co-ordinates in $\mathbf{Z}$ will be found first. If ExactBound is set to true, then points that are found with height larger than B will be ignored.

Example H114E38

```
> P<x,y,z>:=ProjectiveSpace(Rationals(),2);
> C:=Curve(P,x^3+9*y^3+73*z^3);
> time PointsCubicModel(C,10^4);
[ (31/2 : -15/2 : 1), (-353/824 : -1655/824 : 1), (-463/106 : 111/106
: 1), (-1 : -2 : 1), (-2347/526 : 635/526 : 1), (-206/5 : 99/5 : 1),
(22/5 : -13/5 : 1), (-43/16 : -29/16 : 1), (-25 : 12 : 1), (-4493/1076
: -299/1076 : 1), (-215/47 : 64/47 : 1), (-631/151 : -22/151 : 1), (-4
: -1 : 1), (278/193 : -393/193 : 1), (-328/55 : 137/55 : 1), (311/88 :
-207/88 : 1), (145/71 : -148/71 : 1) ]
Time: 2.790
```

114.12.2 Regular Models of Arithmetic Surfaces

Let F be a global field (the rationals, a number field or a function field) with ring of integers O_F. Given a curve C over F with integral defining equations, the associated *arithmetic surface* is the scheme of relative dimension 1 over SpecO_F defined by the same equations. In Magma, one can compute a regular model of the arithmetic surface locally at a given prime of O_F, and extract information from it. The 'model' is a data structure which contains several affine patches with maps between them, as well as the components of the special fibre, and other data. This raw data is quite bulky; the interesting information is accessed via the functions described in this section. This functionality may be expanded (on request) in later versions.

The Magma category for these data structures is `CrvRegModel`.

Caveat: in the initial implementation there are some restrictions on which curves, and which fields, can be handled. These restrictions are not documented here, and may be lifted in the near future.

114.12.2.1 Creation of Regular Models

```
RegularModel(C, P)
```

Verbose `RegularModel` *Maximum :* 2

This computes a regular model of the curve C at the prime P. Here C is a curve over a field F (the rationals, a number field or a univariate rational function field), and P is a prime of the maximal order O_F of F (given as an element or as an ideal). The defining equations of C must have integral coefficients, and the reduction of C modulo P must have dimension 1.

The function returns a model of C: this consists of several affine patches, given by integral equations, which together describe a scheme over O_F whose generic fibre is isomorphic to C. (The gluing maps, and the isomorphism to C, are part of the stored data). The model is regular on the special fibre above P. However it is not necessarily a minimal model.

In some cases, the function may replace F by a finite extension L/F in which P is unramified, and return a regular model for the base change C_L. (This occurs when there is a non-regular point or component in the special fibre that is not geometrically irreducible over the residue field.) When this occurs, a warning message is printed.

114.12.2.2 Using Regular Models

```
IntersectionMatrix(M)
```

Given a regular model, this returns a matrix whose entries are the intersection multiplicities of the (reduced, irreducible) components of the special fibre. Secondly, it returns a sequence giving the multiplicities of the components. (The components of the model always come in the same order.)

```
ComponentGroup(M)
```

Given a regular model of a curve C at a prime P, this returns (as an abstract abelian group) the group of components of the Neron model of the Jacobian of C over the completion at P. (This is computed from the `IntersectionMatrix` of the model.)

```
PointOnRegularModel(M, x)
```

Given a regular model of a curve C over a global field F, and a point $x \in C(F)$, this lifts x to a point on the regular model. More precisely, it finds a patch of the model, and a point on the generic fibre of that patch which maps to x (under the isomorphism between the generic fibres of M and C).

Three objects are returned: a sequence giving coordinates of the point, a sequence containing the equations of the relevant patch, and (for internal use) the 'index' of the patch.

114.12.3 Minimization and Reduction

Minimization and reduction is a search for a linear transformation, that leads to nice equations. The general strategy is described in Section 116.4.4.

Here, we describe the routines for the minimization of plane quartic curves and the reduction for plane curves of degree at least 3. The reduction is done by constructing a cluster of special points on the curve. Thus, we start with this.

```
ReduceCluster(X)
```

`eps`	FldReElt	*Default :* 1e-6
`c`	FldReElt	*Default :* 1
Verbose	`ClusterReduction`	*Maximum :* 3

Here X is a sequence of n-dimensional vectors of complex numbers. The routine returns the cluster in a better embedded form, the transformation matrix applied and its inverse.

Optional arguments: `eps` is the bound used to decide whether a floating point number is zero, and `c` is the initial value of the acceleration factor.

The algorithm is due to Stoll (see [Sto11]).

This routine can be used for reduction of any variety that has a finite and stable set of special points, by using the transformation that reduces the point set.

```
ReducePlaneCurve(f)
```

Here f is a homogeneous polynomial in 3 variables of degree > 2 with integral or rational coefficients. A new polynomial and the transformation matrix applied are retured.

The routine computes the intersection points of f with its hessian. Then the cluster reduction is applied to this point set.

114.12.3.1 Minimization and Reduction for Plane Quartics

```
MinimizePlaneQuartic(f,p)
```

Verbose PlaneQuartic *Maximum :* 1

Given a plane quartic defined by the polynomial f with integer coefficients this routine computes an at p minimized model of the quartic. The new quartic and the transformation used are returned.

```
MinimizeReducePlaneQuartic(f)
```

Verbose PlaneQuartic *Maximum :* 1

Given a smooth plane quartic curve defined by the polynomial f with integer coefficients this routine computes a minimized and reduced model of the curve. The transformation matrix returned applied to f will evaluate to a scalar multiple of the returned polynomial.

For the reduction step, `ReducePlaneCurve` is used.

Example H114E39

Here we test the code by taking a bad embedding and a of curve.

```
> _<x,y,z> := PolynomialRing(Integers(), 3);
> C := -3*x^4 + 7*x^3*y - 2*x^3*z + 6*x^2*y^2 + 9*x^2*y*z - 9*x^2*z^2
>       + 10*x*y^3 - 7*x*y^2*z + 5*x*y*z^2 - 6*x*z^3 - 3*y^4 + 5*y^3*z
>       - 3*y^2*z^2 + 4*y*z^3 + 6*z^4;
> C2 := Evaluate(C,[45*x+346*y,74*y+43*z,62324*z+3462*x]);
> C2;
850482855369981*x^4 - 77028319604430*x^3*y + 61459466820119559*x^3*z -
11625449190228*x^2*y^2 - 4102113209795298*x^2*y*z + 1665400384362332772*x^2*z^2
- 62468022936*x*y^3 - 417499281622764*x*y^2*z - 72808467360772908*x*y*z^2 +
20055880711976359332*x*z^3 - 16293798512*y^4 - 875035770696*y^3*z -
3749491014537304*y^2*z^2 - 430694749052979580*y*z^3 + 90567449117290511049*z^4
> MinimizeReducePlaneQuartic(C2);
6*x^4 - 6*x^3*y + 4*x^3*z - 9*x^2*y^2 + 5*x^2*y*z - 3*x^2*z^2 - 2*x*y^3 +
9*x*y^2*z - 7*x*y*z^2 + 5*x*z^3 - 3*y^4 + 7*y^3*z + 6*y^2*z^2 + 10*y*z^3 - 3*z^4
[    14878   4611976 -21564104]
[    -1935    148866   2804580]
[     3330   -256188   1197852]
```

We do not get the initial curve, but we get a curve with coefficients of the same size.

114.13 Minimal Degree Functions and Plane Models

This section contains functionality to compute smallest degree covering maps from a curve to the projective line $\mathbf{P}^1$, which are equivalent to smallest degree functions on the curve. We refer to such maps as *gonal* maps (the degree of such a map is usually referred to as the *gonality* of the curve). There are intrinsics for all general type (genus ≥ 2) curves of genus less than seven and for all trigonal (gonality 3) curves. The trigonal cases use the Lie algebra method to construct degree 3 maps following the algorithm in [SS]. The 4-gonal cases for genus 5 and 6 curves use the algorithms in [Hara]. Hyperelliptic cases (gonality 2) are dealt with directly using the canonical map and parametrisation of rational curves.

Gonal maps may not exist over the base field and the intrinsics here will return a gonal map over a finite extension in such cases. In some cases, the degree of a minimal extension over which a gonal map may be constructed is determined exactly. In others, this may involve difficult arithmetic problems (e.g., finding a point of minimal degree on a plane sextic over $\mathbf{Q}$) and the extension used may not be of minimal degree. This is discussed further in the descriptions of the intrinsics. The intrinsics are designed for input curves defined over number fields or finite fields.

There are also some intrinsics to compute minimal degree plane models of curves of genus 5 and 6. In the genus 6 case, these birational models may be defined over a small extension field of the base field (an exact minimal degree extension is found here). In the genus 5 case, the intrinsic computes models over the base field given a rational point or divisor of degree 2 in the generic case that may be input by the user. The functionality here is incomplete and doesn't currently cover every type of genus 5 or 6 curve or lower genus curves. We plan to extend this in later releases.

114.13.1 General Functions and Clifford Index One

The algorithm of Schicho and Sevilla used for trigonal curves actually covers the slightly more general case of Clifford index one. An algebraic curve has Clifford index one iff it is trigonal or it is of genus 6 and isomorphic to a non-singular plane quintic. In the latter case, the curve has gonality 4. Classical theory (e.g. Petri's theorem) tells us that a curve of genus greater than one has Clifford index 1 precisely when its canonical image is not defined by quadrics alone. For genus > 3, a minimal basis for the ideal defining the canonical image will then consist of quadrics and cubics. A non-hyperelliptic genus 3 curve is always trigonal and it's canonical image is defined by a smooth quartic in the projective plane.

This gives a simple computational test for Clifford index 1. There is an intrinsic described below that may be called directly by the user for Clifford index 1 canonical curves that returns a gonal map to $\mathbf{P}^1$ in the trigonal case or gives a birational map to a smooth plane quintic.

```
GenusAndCanonicalMap(C)
```

Convenience function for the user. Returns the genus g of the curve C, a Boolean value which is `true` iff $g \leq 1$ or C is hyperelliptic, and the canonical map from C to its canonical image if $g > 1$.

`CliffordIndexOne(C)`

`CliffordIndexOne(C,X)`

The curve C should be a (non-singular) canonical model of a curve of Clifford index 1 (this condition may be tested as described in the introduction). Computes and returns a degree 3 map to the projective line $\mathbf{P}^1$ or a birational map onto a smooth plane quintic, depending on whether C is trigonal or not. In the trigonal case, the map may be defined over a quadratic extension of the base field for curves of even genus. This will occur only if no such map exists over the base field. In the plane quintic case, the map is always defined over the base field. The trigonal case requires that the characteristic of the base field is not 2.

The algorithm used is that of Schicho and Sevilla that applies the Lie algebra method to explicitly compute the fibration map to $\mathbf{P}^1$ of the rational scroll surface X that is defined by the quadrics in the defining ideal of the canonical curve C. The second version of the intrinsic also takes X as an argument in case the user has already computed it.

Example H114E40

```
> P5<x,y,z,s,t,u> := ProjectiveSpace(Rationals(),5);
> C := Curve(P5,[
> -y*z+x*s-y*s+z*t-4*s*t+t^2+2*s*u-3*t*u+2*u^2,
> -z^2+s^2+x*t+2*z*t-2*t^2-4*s*u-2*t*u+4*u^2,
> -z*s-s^2+z*t+t^2+x*u-2*s*u-7*t*u+6*u^2,
> -z*s+s^2+y*t+2*s*t-t^2-2*s*u+t*u,
> -s^2+s*t+y*u-s*u-t*u+u^2,
> -s*t+t^2+z*u-s*u-3*t*u+2*u^2,
> x^3-3*x^2*y+7*x*y^2+3*y^3+x*y*z-y*z^2+9*z^3+3*y^2*s-3*y*z*s-7*z^2*s-7*y*s^2+
> 17*z*s^2-86*s^3-4*z*s*t-16*s^2*t-185*z*t^2+807*s*t^2-406*t^3+17*s^2*u-
> 666*s*t*u+1391*t^2*u+228*s*u^2-1378*t*u^2+393*u^3,
> x^2*z-2*x*y*z+5*y^2*z+y*z^2+3*z^3+8*y^2*s+y*z*s-7*z^2*s+3*y*s^2+17*z*s^2-
> 47*s^3+6*z^2*t+9*z*s*t+s^2*t+32*z*t^2+11*s*t^2-103*t^3-51*s^2*u-171*s*t*u+
> 433*t^2*u+89*s*u^2-577*t*u^2+246*u^3,
> x*z^2-y*z^2-z^3+4*y*z*s+z^2*s+8*y*s^2-3*z*s^2-5*s^3+7*z^2*t-3*z*s*t+17*s^2*t+
> 8*z*t^2-2*s*t^2+62*t^3-41*s^2*u-34*s*t*u-28*t^2*u+14*s*u^2-127*t*u^2+92*u^3
> ]);
> // C a genus 6 canonical curve of gonality 3
> mp3 := CliffordIndexOne(C);
> mp3;
Mapping from: Crv: C to Curve over Rational Field defined by
0
with equations :
x - 2*z + 10*s - 7*t + 10*u
-2/5*x - 24/5*s + 26/5*t - 44/5*u
```

We can check that `mp3` gives a degree 3 map by seeing that the degree 10 linear pencil of divisors generated by D_1 and D_2, the hyperplane sections of C given by the two defining equations of the map, has a common degree 7 factor.

```
> defs := DefiningPolynomials(mp3);
> D1 := Scheme(C,defs[1]);
> D2 := Scheme(C,defs[2]);
> D12 := Scheme(C,defs);
> Degree(D1); Degree(D2);
10
10
> Dimension(D12); Degree(D12);
0
7
```

114.13.2 Small Genus Functions

This section contains intrinsics to compute gonal maps for curves with genus greater than one and less than seven.

`Genus2GonalMap(C)`

Returns a degree 2 map from genus 2 curve C to the projective line. The map is defined over a quadratic extension of the base field k iff no such map exists over k. The map is just the canonical map followed by an inverse parametrisation of its image.

`Genus3GonalMap(C)`

`IsCanonical` BoolElt *Default :* `false`

For a genus 3 curve C, returns the gonality (2 or 3) and a gonal map to $\mathbf{P}^1$. C is trigonal precisely when it is non-hyperelliptic, when its canonical image is a plane quartic Q. Gonal maps are given by the canonical map onto Q followed by projection from a point on Q. Constructing one requires finding a point on Q. If Q is defined over the rationals, a point search up to a small height is applied, and if Q is defined over a finite field a point enumeration is applied to try to find a point over the base field. If this fails to locate a point or if the base field is a number field, a point of Q over an extension of degree ≤ 4 of the base field is used. Thus the map returned may be over an extension of the base field and this extension may not be of minimal degree.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

`Genus4GonalMap(C)`

`IsCanonical` BoolElt *Default :* `false`

For a genus 4 curve C, returns the gonality (2 or 3) and a gonal map to $\mathbf{P}^1$. C is trigonal precisely when it is non-hyperelliptic. In the trigonal case, a gonal map is computed using the `CliffordIndexOne` intrinsic. This involves computing a fibration map for a quadric surface in $\mathbf{P}^3$ containing the canonical image of C, which may be defined over a biquadratic extension of the base field in bad cases. So the map returned may be defined over an extension of degree 4, whereas the map on C should always be defined over an extension of degree at most 2. We will try to fix this in later releases.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

`Genus5GonalMap(C)`

`DataOnly` BoolElt *Default :* `false`

`IsCanonical` BoolElt *Default :* `false`

For a genus 5 curve C, returns the gonality (2, 3 or 4) and a gonal map to $\mathbf{P}^1$. In the gonality 4 case, it also returns some extra data that gives the parametrisation of the set of degree 4 linear pencils (g^1_4s) which give the gonal maps.

The hyperelliptic case (gonality 2) is handled as usual by parametrising the canonical image which gives a gonal map that may be defined over a quadratic extension of the base field.

The trigonal case is easy to deal with here since the rational scroll X that contains the canonical image of C is of codimension 2 in its ambient. We directly compute the fibration map of X from the minimal free polynomial resolution of the canonical coordinate ring of C. This gives a gonal map on C that is always defined over the base field.

In the general (4-gonal case), there are infinitely many equivalence classes of gonal maps which are parametrised by a plane quintic curve F. This along with a function f which takes a point in $F(K)$ as an argument are returned as third and fourth return values. f evaluated at $p \in F(K)$ will return the corresponding gonal map on C which is defined over K or a quadratic extension of it. The actual gonal map returned as the second return value is given by searching for a point on F over a small extesnion of the base field. We use a point search as for `Genus3GonalMap` if the base field is the rationals or a finite field. Otherwise or if this fails, we find a point over an extension field by decomposing a random hyperplane section of F. We try to use singular points on F, if they exist, which correspond to g^1_4s whose associated rational scroll is defined by a rank 3 rather than a rank 4 quadric.

If the parameter `DataOnly` is set to `true` (the default is `false`), then only the gonality is returned when it is less than 4, and only the gonality, F and f are computed and returned in the 4-gonal case.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

`Genus6GonalMap(C)`

`DataOnly` BoolElt *Default :* `false`

`IsCanonical` BoolElt *Default :* `false`

For a genus 6 curve C, returns the gonality (2, 3 or 4), a second type identifer and a gonal map to $\mathbf{P}^1$. The second type number is irrelevant in the gonality 2 and 3 cases, when it is always 1. It is 1,2 or 3 in the 4-gonal case.

For 4-gonal curves of secondary type 2, C is a double cover of a genus 1 curve E through which all gonal maps factor. In this case, the map giving this double covering is returned as a fourth return value. Secondary type 3 curves are isomorphic to (smooth) plane quintics and an isomorphism to such a quintic is returned as the fourth return value. These maps are defined over the base field.

The hyperelliptic case (gonality 2) is handled as for `Genus5GonalMap`.

The trigonal case requires the characteristic of the base field to not be 2. It uses the `CliffordIndexOne` intrinsic and constructs a gonal map that is defined over a quadratic extension of the base field k if no such map exists over k.

There are 3 distinct subcases of the 4-gonal case, which are distinguished by the second return number.

Type 2 curves are double covers of a genus 1 curve E and there are infinitely many 4-gonal maps which are given by composing this double cover with a degree 2 map of E to $\mathbf{P}^1$. E is constructed as a projective normal curve of degree 5 in $\mathbf{P}^4$. The gonal map returned is found by looking for k-rational points on E by general point search methods and, if this fails, taking a point on E defined over a finite extension of k lying in a random hyperplane section of E.

Type 3 curves are (birationally) isomorphic to a plane quintic C_5. The infinitely many 4-gonal maps are given by projection from a point on C_5. The gonal map returned is constructed by finding a k-rational point on C_5 or one over a finite extension as for type 2.

Type 1 (general type) curves have only finitely many gonal maps up to equivalence (5 at most, in fact). The algorithm explicitly finds algebraic data defining these map classes. It chooses one defined over k if possible. Otherwise, we take one over a minimal degree extension of k. The gonal map returned is thus defined over a minimal degree extension of k for such maps.

If the parameter `DataOnly` is set to `true` (the default is `false`), then only the gonality (and secondary type 1) is returned when it is less than 4, and only the gonality, secondary type t and fourth return value when $t = 2$ or 3 are computed and returned in the 4-gonal case.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

Example H114E41

We give some examples with 4-gonal curves of genus 5 and 6. Firstly, we take a random canonical curve of genus 5 given by the intersection of 3 quadrics in $\mathbf{P}^4$.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> C := Curve(P4,[
> -x^2-x*y-y*z+z^2-x*t-y*t-z*t+t^2-x*u+y*u-t*u,
> -x*y+y^2+y*z+x*t+y*t-z*t+z*u+t*u+u^2,
> -x^2-x*y-y*z+z^2+x*t-y*t+t^2-x*u+y*u+z*u+t*u+u^2]);
> g,mp4,F,f := Genus5GonalMap(C);
> g;
4
> mp4;
Mapping from: Crv: C to Curve over Rational Field defined by
0
with equations :
z + u
t
> F;
Curve over Rational Field defined by
7*u^5+6*u^4*v-50*u^3*v^2+40*u^2*v^3+3*u*v^4+2*v^5+48*u^4*w-22*u^3*v*w-
 50*u^2*v^2*w+30*u*v^3*w+10*v^4*w+134*u^3*w^2-24*u^2*v*w^2-36*u*v^2*w^2+
 8*v^3*w^2+187*u^2*w^3+6*u*v*w^3-19*v^2*w^3+129*u*w^4+6*v*w^4+35*w^5
```

Next, we look at a genus 6 curve that is a degree 2 cover of a genus 1 curve.

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> C := Curve(P3,[ x^2*y^2-x^2*t^2-z^2*t^2+2*t^4,
>    y^4-x*z^2*t-2*y^2*t^2+2*x*t^3+t^4,
>    x^3-y^2*t+t^3 ]);
> Genus(C);
6
> g,t,mp4,mpE := Genus6GonalMap(C);
> g; t;
4
2
> mp4;
Mapping from: Crv: C to Curve over Rational Field defined by
0
with equations :
x
t
> mpE;
Mapping from: Crv: C to Curve over Rational Field defined by
$.1^2 + $.2*$.3 - $.4*$.5,
$.1*$.2 + $.3^2 - $.5^2,
$.2^2 - $.1*$.3,
$.2*$.4 - $.1*$.5,
$.3*$.4 - $.2*$.5
```

```
with equations :
x^2
x*t
t^2
x*y
y*t
```

114.13.3 Small Genus Plane Models

This section contains intrinsics to compute minimal degree birational plane models for curves of genus 5 or 6.

`Genus6PlaneCurveModel(C)`

`IsCanonical` BoolElt *Default :* `false`

For a genus 6 curve C of gonality 4 which isn't the double cover of a genus 1 curve (i.e. subtype 2 for `Genus6GonalMap`), returns a birational map to a plane curve of minimal degree (5 or 6). This map may be defined over a finite extension of the base field k, but it will always be a minimal degree extension for the existence of minimal degree plane models. The first return value is a boolean, which is `true` only if C is of gonality 4 and not of subtype 2. If so, the second return value is the map to the curve.

When C is birationally isomorphic to a plane quintic C_5, (subtype 3 for `Genus6GonalMap`), C_5 is a minimal degree plane model and it and the birational map to it are defined over k. This is a Clifford index 1 case, and the computation is performed via the `CliffordIndexOne` intrinsic.

In the general case, (subtype 1 for `Genus6GonalMap`), the smallest degree plane models are of degree 6, there are only finitely many birational maps from C to degree 6 plane curves up to linear equivalence and these are in 1-1 correspondence with the finitely many equivalence classes of gonal maps. The birational maps to plane models are computed by a slight variant of the same algorithm used to compute the gonal maps (see [Hara]). The return value is over the base field k, if possible, and otherwise over a smallest degree (at most 5) possible field extension.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

`Genus5PlaneCurveModel(C)`

`Genus5PlaneCurveModel(C,P)`

`Genus5PlaneCurveModel(C,Z)`

`IsCanonical` BoolElt *Default :* `false`

For a genus 5 curve C, tries to compute a birational map over the base field k to a plane curve of minimal degree (5 or 6) for plane curve models over $\bar{k}$. The first return value is a boolean which is `true` if the construction succeeds and, if so, the second return value is the birational map to the plane curve.

If C is hyperelliptic, the return value is always `false`. Gonality 3 curves are always birational over k to a plane curve of degree 5 with a single singular point and the computation always succeeds. The map to $\mathbf{P}^2$ on a canonical model C_c of C lying in $\mathbf{P}^4$ is given by projection from a line lying in the rational scroll surface X that contains C_c.

If C is 4-gonal and isn't a double cover of a genus 1 curve, the minimal degree plane model is of degree 6 and there are infinitely many of these (up to linear equivalence) that are given by projection from a secant line or tangent line of the canonical model C_c.

To find a birational map to such a model over k, we need a secant or tangent line defined over k, which will correspond to a k-rational point on C_c or a k-rational reduced divisor of degree 2 (i.e. 2 distinct points that are k-rational or conjugate over a quadratic extension of k).

The second and third versions of the function have a second argument that should be a k-rational *non-singular* point on C or a reduced subscheme of C of dimension 0 and degree 2, whose points (over $\bar{k}$) lie in the non-singular locus of C. These are used to define the k-rational tangent line or secant line on the canonical model and the intrinsic will always succeed and return the map to a degree 6 model if they are provided by the user.

In the version with only the curve C as input, the intrinsic attempts to find a (non-singular) k-rational point on C using `PointSearch` if the base field is the rationals or using point enumeration if the base field is a finite field. If this search fails, or k is a field of a different type, the intrinsic will fail and return `false`.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

Example H114E42

We use the intrinsic for genus 6 curves to get a degree 6 plane model for the modular curve $X_0(58)$, starting from its canonical model.

```
> X := X0NQuotient(58,[]);
> X;
Curve over Rational Field defined by
x[1]^2-x[1]*x[3]+x[2]*x[4]+x[2]*x[5]-x[1]*x[6]+x[4]*x[6]+x[5]*x[6],
x[1]^2-x[1]*x[2]-x[2]^2-x[1]*x[3]-x[1]*x[4]+x[3]*x[4]+x[2]*x[5]+x[3]*x[5],
-x[1]^2+x[1]*x[2]+x[2]^2+x[4]^2+x[4]*x[5]-x[2]*x[6],
-x[1]^2-x[1]*x[2]+x[2]^2+x[2]*x[3]-x[1]*x[4]+x[2]*x[4]+x[3]*x[4]+x[4]^2-x[5]*x[6],
x[2]^2-x[1]*x[3]+x[2]*x[3]+x[3]^2-x[1]*x[4]+x[3]*x[4]+x[2]*x[6]+x[3]*x[6],
x[1]*x[2]-x[2]*x[3]-x[1]*x[4]+x[3]*x[4]+x[2]*x[5]+x[3]*x[5]+x[4]*x[5]-x[1]*x[6]+
x[2]*x[6]+x[3]*x[6]+x[4]*x[6]
> boo,mp := Genus6PlaneCurveModel(X : IsCanonical := true);
> boo;
```

```
true
> C<x,y,z> := Codomain(mp); //the plane model
> C;
Curve over Rational Field defined by
x^6-3*x^5*y+x^4*y^2+3*x^3*y^3-7/4*x^2*y^4-1/4*x*y^5-1/4*y^6+3/2*x^5*z-x^4*y*z-
9/2*x^3*y^2*z+7/2*x^2*y^3*z+5/8*x*y^4*z+3/4*y^5*z+5/4*x^4*z^2+1/4*x^3*y*z^2-
3*x^2*y^2*z^2+3/4*x*y^3*z^2-3/4*y^4*z^2+5/8*x^3*z^3+5/4*x^2*y*z^3-7/4*x*y^2*z^3+
1/4*y^3*z^3+1/8*x^2*z^4+5/8*x*y*z^4-1/8*y^2*z^4+1/8*y*z^5
> mp;
Mapping from: Crv: X to Crv: C
with equations :
x[1] - x[3]
x[2] + x[6]
x[5] + x[6]
```

Do the same for genus 5 4-gonal modular curve $X_0(42)$.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> X42 := Curve(P4,[x*z+z^2+x*t-x*u,
>   y^2-2*y*z+z^2-x*t-2*y*t+z*t-2*y*u+z*u+t*u,
>   x^2+x*t+y*t+y*u ]);
> // use pointsearch to pick rational point on X42
> boo,mp := Genus5PlaneCurveModel(X42 : IsCanonical := true);
> boo;
true
> C<x,y,z> := Codomain(mp);
> C;
x^6+3*x^5*y+9/2*x^4*y^2+2*x^3*y^3+2*x^2*y^4+11/8*x^5*z+31/8*x^4*y*z+23/4*x^3*y^2*z
+1/2*x^2*y^3*z+3*x*y^4*z+85/128*x^4*z^2+29/16*x^3*y*z^2+41/16*x^2*y^2*z^2-
9/4*x*y^3*z^2+9/8*y^4*z^2+9/64*x^3*z^3+17/32*x^2*y*z^3+15/16*x*y^2*z^3-
9/8*y^3*z^3+1/128*x^2*z^4+3/32*x*y*z^4+9/32*y^2*z^4
> mp;
Mapping from: Crv: X42 to Crv: C
with equations :
x + 1/2*t - 1/2*u
y - 1/4*t + 1/4*u
z - t + u
> // use nicer chosen point for nicer map
> boo,mp := Genus5PlaneCurveModel(X42,X42![0,0,0,0,1] : IsCanonical := true);
> C<x,y,z> := Codomain(mp);
> C;
x^5*y-2*x^4*y^2+x^3*y^3-x^2*y^4+2*x*y^5-y^6-x^5*z+3*x^4*y*z-6*x^3*y^2*z-
11*x^2*y^3*z-12*x*y^4*z-2*x^4*z^2+9*x^3*y*z^2+20*x^2*y^2*z^2+16*x*y^3*z^2+
2*y^4*z^2-4*x^3*z^3-7*x^2*y*z^3-4*x*y^2*z^3-x^2*z^4-2*x*y*z^4-y^2*z^4
> mp;
Mapping from: Crv: X42 to Crv: C
with equations :
x
y
```

```
z + t
```

114.14 Bibliography

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**Die06**] Claus Diem. An Index Calculus Algorithm for Plane Curves of S mall Degree. In Hess et al. [HPP06], pages 543–557.

[**Elk00**] N. Elkies. Rational Points Near Curves and Small Nonzero $|x^3 - y^2|$ via Lattice Reduction. In Bosma [Bos00], pages 33–63.

[**Ful69**] William Fulton. *Algebraic Curves.* Mathematics Lecture Note Series. W. A. Benjamin, New York-Amsterdam, 1969.

[**Har**] M. C. Harrison. Explicit solution by radicals, gonal maps and plane models of algebraic curves of genus 5 or 6. Preprint: online at arXiv as arXiv:1103.4946v3[math. AG].

[**Har77**] Robin Hartshorne. *Algebraic Geometry, GTM 52.* Springer, ASpringer, 1977.

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

[**HS10**] J. Hilmar and C. Smyth. Euclid Meets Bézout: Intersecting Algebraic Plane Curves with the Euclidean Algorithm. *The American Mathematical Monthly*, 117:250–260, (March) 2010.

[**SS**] J. Schicho and D. Sevilla. Effective radical parametrization of trigonal curves. Preprint: online at arXiv as arXiv:1104.2470v1[math.AG].

[**ST02**] Frank-Olaf Schreyer and Fabio Tonoli. Needles in a Haystack: Special Varieties via Small Fields. In Eisenbud et al., editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of *Springer Algorithms and Computation in Mathematics Series*, pages 251–277. Springer-Verlag, 2002.

[**Sto11**] Michael Stoll. Reduction theory of point clusters in projective space. 2011.

[**vLvdG88**] J. H. van Lint and G. van der Geer. *Introduction to Coding Theory and Algebraic Geometry*, volume 12 of *DMV Seminar.* Birkhaeuser, Basel, 1988.

115 RESOLUTION GRAPHS AND SPLICE DIAGRAMS

Chapter 115
RESOLUTION GRAPHS AND SPLICE DIAGRAMS

115.1 Introduction

Both resolution graphs and splice diagrams are labelled graph-like diagrams used to encode geometric data closely related to some resolution of singularities procedure in algebraic geometry. They are commonly used to visualise this data. Of course, there are other tools in Magma, Puiseux expansions for instance, which can be used if preferred. A typical example is when a configuration of curves on a surface is the given data. In this case, *dual graph* of the configuration has vertices corresponding to the individual curves and edges corresponding to their intersections. The vertices may be labelled with the selfintersections of the corresponding curves and possibly also with the multiplicities with which the curves appear in the configuration.

This chapter discusses two different enhanced graph types: `GrphRes` for resolution graphs and `GrphSpl` for splice diagrams. Neither of them is literally a graph in Magma; in particular, functions taking graphs as argument cannot be applied directly to objects defined here. Instead they work by having a directed graph, referred to as the *underlying graph*, as a primary attribute and by caching other data (which is typically associated to particular vertices and edges of the graph) in sequences as secondary attributes. There are also vertex types which allow the convenient idiom of Magma's graph package to be used. Note, however, that unlike other graph types, these do not have edge types.

Graph surgery routines cannot be used directly since they must manage both the underlying graph and the associated data. A collection of appropriate surgery functions, those used in resolution routines, have been provided in this context; they are usually brief, simply concatenating attribute data whenever it is present on both sides.

Functions to recover data related to the graph, whether globally or at a particular vertex or edge, are also provided. Thus the user does not access labels of the graph but rather uses the intrinsics listed later in this chapter. Of course, usually these do no more than unload an attribute.

115.2 Resolution Graphs

A resolution graph g is a graph with data associated to its vertices. The underlying graph is the dual graph of a blowup process. The vertices correspond to rational curves E, "exceptional curves", on some surface S which are contracted by some map $f : S \to P^2$, where P^2 is the projective plane. Such maps arise during the resolution process by blowups of a curve singularity, say $p \in C$ where C is a curve in P^2. In that context, C can be pulled back to S using f and the pullback can be decomposed (as an effective divisor on S) as

$$f^*C = \widetilde{C} + E_C$$

where $\widetilde{C}$ is the *birational transform* of C and E_C is an effective divisor supported on the union of the exceptional curves. The surface S is never realised as a geometric object.

For a vertex v of g, the corresponding rational curve is often denoted E_v. The edges of g correspond to the intersections between different E: vertices v and w are joined by edges corresponding in some way to the points of intersection of the curves E_v and E_w. In the contexts used below, any two curves will meet at most in one point and that will be a transverse intersection. So the intersection number E_vE_w on S will be the number of edges between v and w, either zero or one, if they are distinct vertices and a selfintersection number associated to v otherwise. Thus the graph enables basic intersection calculations to be carried out implicitly on S.

The data associated to each vertex v is the following where $E = E_v$: the selfintersection E^2; the coefficient of E in some pullback divisor, often E_C but this depends on the context; the coefficient of E in a representative of the canonical class of S locally supported on the exceptional curves; the number of transverse intersections of the birational transform $\widetilde{C}$ of C with E. At each vertex v, this data is often denoted s_v, m_v, k_v, t_v respectively. This is the coarsest kind data one could want. It enables basic intersection theory calculations (including calculating the contribution of a singularity p of a curve C to the genus of C). For more detailed calculations, the map f and the equation of the birational transform of C are held on patches along each E. In each case, the patch is an affine xy-plane with E as the line $y = 0$ in it (although it is actually the closure of this patch and map that is recorded).

Resolution graphs are usually created implicitly by some geometrical algorithm like the resolution of a plane curve singularity. The first creation methods below are of this nature. But graphs can also be created explicitly by providing the required data. How much data is required varies according to the purpose of the graph; some common alternatives are included here.

115.2.1 Graphs, Vertices and Printing

Resolution graphs are less complicated as a type than the other graphs in Magma. They do not have associated vertex and edge sets. However, there is a resolution graph vertex type so that vertices can be passed between intrinsics.

Graph printing is similar to that of the underlying directed graph. An example is

```
The resolution graph on the Digraph
Vertex  Neighbours
1 ([ -2, 9, 1, 0 ])     2 ;
2 ([ -4, 18, 2, 0 ])    3 ;
3 ([ -2, 63, 9, 0 ])    4 6 ;
4 ([ -2, 42, 6, 0 ])    5 ;
5 ([ -2, 21, 3, 0 ])    ;
```

```
6 ([ -1, 66, 10, 3 ])   ;
```

This is a graph on 6 vertices — they are listed as one of the integers $1, \dots, 6$ down the left. The integer corresponding to a vertex is called the *index* of the vertex. In brackets by each vertex v is a label of the form $[s, m, k, t]$ where s is the selfintersection, m is the multiplicity (the interpretation of which is dependent on the context), k is the canonical multiplicity and t is the number of transverse intersections at v as described in the introduction. Next comes a space-separated list of the vertices at the far end of edges directed away from v. Until one is used to reading graphs in this way, and also even then, drawing the graph by hand is recommended.

The vertex labels can be shorter if some data is missing. The alternatives are $[s]$, $[s, t]$ and $[s, m, t]$ in the previous notation. The most data consistent with what has been calculated for the graph will be displayed.

Notice that, although these really are printed as graph labels, the data in them should not be accessed as such. These labels are often unassigned, or assigned in an unexpected way. They are only intended for printing. There are dedicated functions below for retrieving data associated to a graph.

```
g eq h
```

Returns `true` if and only if the resolution graphs g and h are the same object in Magma.

```
ResolutionGraphVertex(g,i)
```

```
g ! i
```

The vertex of the resolution graph g with index i.

```
Vertex(v)
```

The underlying directed graph vertex of the resolution graph vertex v.

```
ResolutionGraph(v)
```

The resolution graph of which v is a vertex.

```
IsVertex(g,v)
```

Returns `true` if and only if v is a vertex of the resolution graph g.

```
Index(v)
```

The index of the underlying graph vertex of the resolution graph vertex v; this is the integer appearing as the vertex identifier when the graph is printed.

```
v eq w
```

Returns `true` if and only if the resolution graph vertices v and w are the same object in Magma.

115.2.2 Creation from Curve Singularities

Let C be a reduced plane curve, either affine or projective. See Chapter 114 for details of how to create such a curve. Let p be a singular point of C. The intrinsic below calculates the dual graph of the resolution of p on C together with some auxiliary data. As in the introduction, the sequence of blowups required in the resolution is thought of as a morphism $f : S \to P^2$ of projective surfaces; P^2 is the projective plane (which is the closure of the ambient plane of C if it is affine) while S is a surface not realised explicitly in Magma.

The target resolution is the minimum transverse resolution, sometimes called the log resolution, a resolution in which the birational transform $\widetilde{C}$ of C on S is nonsingular and transverse to the exceptional locus. However, there are circumstances under which a larger resolution will be calculated. This makes no difference to the geometric data arising from the resolution.

```
ResolutionGraph(C, p)
```

M	RngIntElt	*Default :* 1
K	RngIntElt	*Default :* 1

```
ResolutionGraph(C, p)
```

Calculate a transverse resolution graph of the plane curve singularity of C at the point p. If the point argument is missing and C is affine, the resolution is calculated at the origin, but in that case parameters cannot be assigned and take the default values.

The numerical parameters determine whether additional data is calculated: value 1, the default, enables the calculation while value 0 omits it.

The parameter M refers to the pullback multiplicities of C. It returns a sequence $[m_v]$ of rational numbers (although always integral in the current algorithms) corresponding to the vertices of the graph. These numbers have the following meaning: on the blownup surface S, as divisors,

$$f^*C = \widetilde{C} + \sum m_v E_v$$

where the sum is taken over the vertices v of the graph and E_v is the corresponding exceptional curve.

The parameter K refers to the canonical multiplicities along g. It returns a sequence $[k_v]$ of rational numbers (although again always integral) corresponding to the vertices of the graph. On the blownup surface S in a neighbourhood of the union of exceptional curves E the canonical class K_S has a representative

$$K_S = \sum k_v E_v$$

where the sum is taken over the vertices of the graph.

The surface S is covered by affine plane patches. The map f to the projective plane can be calculated when restricted to these patches. In fact, the closure of these maps is calculated as a birational automorphism of the projective plane. All blowup algorithms arrange that the exceptional curve E_v corresponding to a vertex v is (a patch on) L_y, the second coordinate line "$y = 0$", in the projective plane. So pulling C back to the plane using the patch map at v will produce a curve having $m_v L_y$ as a component.

The calculation of maps and pullbacks is expensive so some precautions are taken. First, the maps are only calculated at significant vertices of the graph: significant here means that the blowup procedure branches at that vertex, or that $\widetilde{C}$ meets the exceptional locus there. Second, the maps are calculated as a sequence of maps: the map required is the composition of these. The composition will be carried out automatically if needed. (In the current code, this can only be carried out if no field extensions have been made. This will be updated in due course.)

The calculation is carried out by tying together strings of blowups (done recursively in one go using a standard Newton polygon argument). The Newton polygon argument used automatically makes a curve transverse to all axes, not just to exceptional curves. These extra blowups do not invalidate numerical calculations made with the resolution graph (since minimality is never a condition) and they are essential when resolving irregular fibres of pencils.

Example H115E1

First make a curve with a singularity. The following curve demonstrates the typical way singularities with more than one Puiseux pair — those which need the Newton algorithm to be repeated when calculating a resolution or local parametrisation — arise.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,(x^2 - y^3)^2 + x*y^6);
```

The interesting singularity is at the origin. The next two lines calculate the resolution graph of this singularity and display it.

```
> g := ResolutionGraph(C,Origin(A));
> g;
The resolution graph on the Digraph
Vertex  Neighbours
1 ([ -3, 4, 1, 0 ])     2 ;
2 ([ -2, 12, 4, 0 ])    3 4 ;
3 ([ -2, 6, 2, 0 ])     ;
4 ([ -3, 14, 5, 0 ])    5 ;
5 ([ -1, 30, 12, 1 ])   6 ;
6 ([ -2, 15, 6, 0 ])    ;
```

The resulting graph has 6 vertices. So the transverse resolution of this singularity is achieved by 6 blowups. The order of blowup can be determined by the third column, the canonical class: this number strictly increases throughout the process, so curve E_1 is the first exceptional curve, E_3 is the second and so on until finally E_5 is extracted. This could also be deduced from the sequence of selfintersections, the first column. See how the birational image of C after blowing up only intersects E_5, and then only in a single transverse point. In other words, C has a single place at the origin.

115.2.3 Creation from Pencils

Let P be a jacobian pencil in the affine plane A^2, that is, a pencil of the form $f(x, y) = c$ as c varies. See Chapter 112 for details of how to create jacobian pencils. The resolution graph which can be created automatically is the regular resolution graph: that is, the minimal sequence of transverse blowups which resolve the rational map determined by P from the projective plane (the closure of A^2) to the projective line. However, other resolution graphs can be constructed using more explicit creation functions.

```
ResolutionGraph(P)
```

The resolution at infinity of the jacobian pencil $P : f = c$ of some polynomial f as the value c varies defined on some affine space A. This resolution graph is thought of as being rooted with root vertex corresponding to the line at infinity of A. The multiplicities are those of the fibre of f at infinity.

```
ResolutionGraph(P,a,b)
```

The resolution graph at infinity of the union of the two fibres of P above a and b. The multiplicities and canonical class are not calculated automatically for this graph.

Example H115E2

The following is a simple example of a pencil exhibiting interesting behaviour at infinity. It is taken from [Neu99].
First a pencil in some affine plane is made.

```
> A<x,y> := AffineSpace(Rationals(),2);
> f := x^2*y - x;
> P := Pencil(A,f);
> P;
The pencil defined by x^2*y - x
```

Then the regular resolution graph at infinity is calculated.

```
> ResolutionGraph(P);
The resolution graph on the Digraph
Vertex  Neighbours
1 ([ -1, 3, -3, 0 ])    2 5 ;
2 ([ -3, 1, -2, 0 ])    3 ;
3 ([ -1, 0, -2, 1 ])    4 ;
4 ([ -2, 0, -1, 0 ])    ;
5 ([ -2, 2, -2, 0 ])    6 ;
6 ([ -2, 1, -1, 0 ])    7 ;
7 ([ -1, 0, 0, 1 ])     ;
```

The resulting graph has 7 vertices. Vertex 1 corresponds to the line at infinity of the plane, or more properly to its birational image after blowing up. The pencil meets this line at two points,

each of which have undergone three blowups in the resolution process. The vertex labels carry auxiliary data. The first column lists the selfintersections of the curves E_v corresponding to the vertices v. The general fibre of the pencil meets a single exceptional curve above each of the points as shown by the fourth column. The fibre at infinity is $3E_1 + E_2 + 2E_5 + E_6$ as shown in the second column. The third column gives the multiplicities of a canonical divisor on the blownup surface: K_S can be represented by the divisor $-3E_1 - 2E_2 - 2E_3 - E_4 - 2E_5 - E_6$.
It is known that the pencil P is irregular at infinity above 0. This can be seen by calculating the explicit resolution of the union of two fibres, $f = 0$ and a general one, say $f = 1$.

```
> ResolutionGraph(P,0,1);
The resolution graph on the Digraph
Vertex  Neighbours
1 ([ -1, 0 ])   2 5 ;
2 ([ -3, 0 ])   3 ;
3 ([ -1, 1 ])   4 ;
4 ([ -2, 2 ])   ;
5 ([ -2, 0 ])   6 ;
6 ([ -2, 0 ])   7 ;
7 ([ -1, 2 ])   ;
```

The multiplicities and canonical class have not been calculated by this function. Later in this chapter there are functions which will do this, although the multiplicities require some interpretation. The selfintersections have been calculated as before (and notice that the blowups at least appear to be exactly those of the previous resolution procedure; in fact they are indeed the same). The number of transverse intersections are now those of the union of the two fibres. Since the fibre above 1 is general it contributes one intersection at vertex 3 and one at vertex 7. So the intersections of the irregular fibre can be deduced, namely two at vertex 4 and one at vertex 7.

115.2.4 Creation by Hand

A resolution graph can be created by hand. This can be fiddly if the underlying graph is complicated. See Chapter 149 for details on how to create a directed graph.

```
MakeResolutionGraph(g,s,t)
```

```
MakeResolutionGraph(g,s)
```

The resolution graph on underlying directed graph g. Although it is not checked, the graph g should usually be a directed tree otherwise some reduction algorithms which might be invoked later might not work. In that case, moreover, its root must be the vertex of index 1.

The selfintersections of vertices correspond to the integer entries of the sequence s. If used, the number of transverse intersections of the putative resolved curve with each vertex correspond to the integer entries of the sequence t.

```
MakeResolutionGraph(N)
```

The resolution graph corresponding to the Newton polygon N.

```
UnderlyingGraph(g)
```

The underlying directed graph of the resolution graph g.

115.2.5 Modifying Resolution Graphs

Any resolution graph, whether created by hand or not, can have its numerical data calculated or modified. There are also some functions for performing surgery on the underlying graph.

There are also functions which do the linear algebra calculations typical of the numerical calculations associated with resolution graphs. But beware: they each base their calculation on some part of the data of the graph but make no check that all numerical data is consistent at the end of calculation.

```
Connect(v,w)
```

If v and w are vertices of distinct resolution graphs, return the graph comprising the union of these graphs joined by an edge from v to w. Selfintersections are inherited by the graph from its two components. Multiplicities, canonical class and transverse intersections will be inherited if calculated on both components.

```
CalculateCanonicalClass(~g)
```

Calculate the canonical class supported on the resolution graph g using the selfintersections of the E_v and the assumption that the E_v are nonsingular rational curves meeting transversely. Note that this calculation uses only the selfintersections already associated to g.

```
Disconnect(v,w)
```

If v and w are vertices of a resolution graph g, return the resolution graph with any edge joining them removed. The resulting graph may well be disconnected. The only data preserved is the selfintersections and transverse intersections.

```
Component(v)
```

The connected component of the resolution graph containing vertex v.

```
CalculateMultiplicities(~g)
```

Calculate the pullback multiplicities of the resolution graph g using the selfintersections of the E_v, the assumption that the E_v are nonsingular rational curves meeting transversely, and the number of transverse intersections of $\widetilde{C}$ with the E_v. It is assumed that g is the resolution graph of a curve singularity during this calculation, although that need not be the case. In general there will be some choice of multiplicities. If g is the resolution of a curve singularity, and if the multiplicity of that singularity is cached, then the correct multiplicities can be identified: the multiplicity along the first blownup curve is the same as that of the singularity; the curve which was blown up first can be identified since it alone has canonical multiplicity 1.

However, if g arose as the resolution at infinity of two fibres of a pencil, for instance, the multiplicities calculated here would depend on which two fibres were used. The result would be the unique divisor supported on g which when added to the birational transform of the two affine fibre patches was linearly equivalent to the zero divisor. In particular, if the two fibres were general, the calculation here would return -2 times the fibre at infinity. Otherwise the calculation will return a combination of that and the exceptional components of irregular finite fibres. The intrinsic which calculates the regular resolution graph of a pencil already takes this into account.

```
CalculateTransverseIntersections(~g)
```

Calculate the number of transverse intersections of $\widetilde{C}$ with each E_v on the basis of their selfintersection numbers and multiplicities in the resolution graph g.

```
ModifySelfintersection(~v,n)
```

Change the selfintersection at vertex v of a resolution graph to n.

```
ModifyTransverseIntersection(~v,n)
```

Change the number of transverse intersections at vertex v of a resolution graph to n.

115.2.6 Numerical Data Associated to a Graph

The meaning of the data given here depends on the context in which the graph was created. The case already discussed of a configuration of rational curves arising from the resolution of a curve singularity is the prototype.

Many of these functions can also be applied to a single vertex of a graph: `Selfintersection`, `CanonicalMultiplicity` and so on.

```
Size(g)
```

The number of vertices of the underlying graph of the resolution graph g. Typically, this is the number of exceptional curves in the resolution.

```
SelfIntersections(g)
```

The selfintersections of the vertices of the resolution graph g.

```
Multiplicities(g)
```

The multiplicities of the vertices of the resolution graph g in some divisor.

```
CanonicalClass(g)
```

The multiplicities of the vertices of the resolution graph g in a local representative of the canonical class.

```
TransverseIntersections(g)
```

The number of transverse intersections of some curve (usually used to create g) with the vertices of the resolution graph g.

```
GenusContribution(g)
```

The contribution to the genus of a plane curve of a singularity having g as its resolution graph.

```
CartanMatrix(g)
```

The incidence matrix of the (undirected) graph underlying the resolution graph g with selfintersections on the diagonal.

```
Determinant(g)
```

The determinant of the Cartan matrix of the resolution graph g.

115.3 Splice Diagrams

Splice diagrams are graphs decorated with integer labels at each end of each edge and a number of arrows, often none at all, attached to each vertex. They are fully described in [EN85]. No description of the meaning of splice diagrams will be given here, only the functions that Magma has for manipulating them. (There are two common features of splice diagrams that cannot yet be realised. First, omitting edge labels which are 1 is common but is not allowed. Second, and less trivially, arrow weights are not allowed.

Other data that is also stored with the diagram are vertex multiplicities, canonical multiplicities, and total linking numbers of vertices. The distinction between a splice diagram and its underlying directed graph, as well as that between a splice diagram vertex and its underlying vertex, is often left implicit.

115.3.1 Creation of Splice Diagrams

Splice diagrams are usually created from geometric objects like plane curve singularities or jacobian pencils. They can also be built explicitly by hand. The section on translations between graphs describes functions which create splice diagrams associated to resolution graphs.

```
SpliceDiagram(C,p)
```

The splice diagram of the plane curve singularity of C at the point p.

```
RegularSpliceDiagram(P)
```

The regular splice diagram at infinity of the jacobian pencil P. This diagram is considered to be rooted at vertex 1 corresponding to the line at infinity of the ambient plane of P. Its underlying graph is directed away from this root.

`MakeSpliceDiagram(g,e,a)`

A splice diagram on directed graph g. The edge labels are taken from the sequence e. The elements of this sequence are sequences of pairs of integers $[a, b]$. The i-th element of e will be assigned to the i-th edge of g (in the order returned by `Edges(g)`) with a as the near label and b as the far label (with respect to the directions of the edges of g). The number of arrows at each vertex are taken from the sequence of integers a.

`MakeSpliceDiagram(e,l,a)`

The splice diagram described by the data in the sequences e, l, a. The first two contain sequences of two integers: e is interpreted as a set of directed edges, the integers appearing in it being the vertex indices of the resulting graph. Edge labels are determined by the sequence l as in the previous function. The number of arrows at each vertex are taken from the sequence of integers a.

`SpliceDiagramVertex(s,i)`

The vertex of the splice diagram s having index i.

`SpliceDiagram(v)`

The splice diagram containing the vertex v.

`UnderlyingGraph(s)`

The underlying directed graph of the splice diagram s.

`UnderlyingVertex(v)`

`Vertex(v)`

The vertex of the underlying graph corresponding to the vertex v of a splice diagram.

`Vertices(s)`

The vertices of the splice diagram s.

`RootVertex(s)`

The root vertex of the splice diagram s as a rooted tree directed away from its root by the directions on the edges.

`Index(v)`

The index of the vertex v of a splice diagram.

`s eq t`

`v eq w`

Returns `true` if and only if the splice diagrams s and t are the same object in Magma. This can also be applied to a pair of vertices of a splice diagram.

115.3.2 Numerical Functions of Splice Diagrams

The raw numerical data of a splice diagram is the collection of labels on its edges and vertices together with data recovered from the underlying graph. First we list functions to get hold of this data, and then some more substantial functions which derive many of the natural conclusions of this data, linking number, euler characteristic and so on.

```
EdgeLabels(s)
```

The integer labels on the edges of the splice diagram s.

```
VertexLabels(s)
```

The integer labels on the vertices of the splice diagram s.

```
TotalLinking(v)
```

The total linking number of the vertex v of a splice diagram.

```
LinkingNumbers(s)
```

The total linking numbers of the vertices of the splice diagram s.

```
Linking(u,v)
```

The linking number of vertices u and v of a splice diagram.

```
EdgeDeterminant(u,v)
```

The edge determinant of the edge joining the vertex u to the vertex v of a splice diagram.

```
Valency(v)
```

The splice valency of the vertex v of a splice diagram. This is the valency of v in the underlying graph plus the number of arrows at v.

```
IsRegular(s)
```

Returns `true` if and only if the splice diagram s is regular.

```
IsReduced(s)
```

Returns `true` if and only if the splice diagram s is reduced, that is, it has no valency 2 nodes and no weight 1 leaves.

```
HasIrregularFibres(s)
```

Returns `true` if and only if the splice diagram s has a vertex with zero linking number.

```
Degree(s)
```

The linking number of the first vertex of the splice diagram s.

```
EulerCharacteristic(s)
```

The Euler characteristic of the splice diagram s.

```
Size(s)
```

The number of vertices of the splice diagram s.

```
Arrows(s)
```

A sequence of integers containing the number of arrows of the splice diagram s: the i-th sequence entry is the number of arrows at the vertex of index i. This function can also be applied to a single vertex returning a single integer.

```
VertexPath(u,v)
```

A sequence of vertices on the path from the vertex u to the vertex v of a splice diagram. The second return value is the sequence of products of off-path weights at each vertex.

115.4 Translation Between Graphs

Splice diagrams arise from resolution graphs by a reduction procedure and conversely resolution graphs arise from splice diagrams by a continued fraction calculation. At present, Magma only incorporates the former calculation. However, when a splice diagram s has been constructed using a curve singularity, the corresponding resolution graph g is calculated and can be recovered using the function `CorrespondingResolutionGraph`. The vertices of s correspond to a subset of those of g. The correspondence can be recovered with the function `CorrespondingVertices`.

115.4.1 Splice Diagrams from Resolution Graphs

By default Magma always makes the reduced splice diagram since otherwise many determinants would be calculated unnecessarily.

The translation of resolution graph g to splice diagram is done in two steps. First the underlying graph of g is reduced by the removal of all vertices of valency 2 (including arrows in the valency calculation). Then the edge labels are calculated using determinants of subgraphs.

```
SpliceDiagram(g)
```

`L`	RngIntElt	*Default :* 0
`K`	RngIntElt	*Default :* 0
`Reduced`	RngIntElt	*Default :* 1

A splice diagram of the resolution graph g. All parameters can take the value 0 or 1. If `Reduced` is 1 then the splice diagram will be reduced, otherwise it will be the splice diagram on the underlying graph of g.

The parameter `L` refers to the total linking numbers of the vertices of the resulting splice diagram. The parameter `K` refers to the canonical class of the vertices of the resulting splice diagram. Each quantity will be calculated when the corresponding parameter is 1.

```
SpliceDiagram(g,v)
```

The splice diagram of the resolution graph g with the condition that the vertex v will not be removed by a reduction.

115.5 Bibliography

[**EN85**] D. Eisenbud and W.D. Neumann. *Three-dimensional link theory and invariants of plane curve singularities*, volume 110 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, NJ, 1985.

[**Neu99**] W.D. Neumann. Irregular links at infinity of complex affine plane curves. *Quart. J. Math. Ox. Ser. (2)*, 50:301–320, 1999.

116 ALGEBRAIC SURFACES

Chapter 116
ALGEBRAIC SURFACES

116.1 Introduction

This chapter contains Magma geometric functionality for working specifically with algebraic surfaces and specialised subtypes. We hope to greatly expand this area in upcoming years. The general surface type `Srfc` is for 2-dimensional algebraic varieties over a field (i.e. schemes that are geometrically reduced and irreducible).

The current functionality is largely split between that for (singular) hypersurfaces in ordinary projective 3-space, which is older and relies heavily on the formal desingularisation package of Tobias Beck, and for ordinary projective surfaces in arbitrary dimensional ambients that are "almost non-singular" which relies more on Magma's coherent sheaf package. Almost non-singular here means with only simple (A-D-E) singularities, which are terminal singularities that don't affect the calculations involving the pluri-canonical sheaves. It is useful to allow simple singularities as they naturally occur in a number of models (anticanonically-embedded degenerate Del Pezzo surfaces, minimal models for surfaces of general type).

The surface type `Srfc` is a subtype of the scheme type `Sch` so the general functionality for schemes (see Chapter 112) and coherent sheaves (see Chapter 113) is of course also available.

The first section of the chapter deals with functions for surfaces in general ambients although, as noted above, there are singularity assumptions and the restriction to ordinary projective surfaces for many of the intrinsics.

The next section deals with surfaces in $\mathbf{P}^3$ with no singularity assumptions. this contains functions to compute formal desingularizations, general adjoint linear systems, classification and reduction of rational surfaces to special type and a general parametrization routine.

The final section deals with specific code for Del Pezzo surfaces, the specialised subtype for which we currently have the most functionality. There are parametrization routines over the rationals (or number fields in some cases) for Del Pezzos by degree and constructions for degree 6 Del Pezzos associated to different twisted torus type. There are also minimisation and reduction routines for degree 3 and 4 Del Pezzos and construction, point-counting and computation of invariants for degree 3.

The biggest problems here are the singularity assumptions and restrictions to ordinary projective space which means that large dimensional ambients are unavoidable at times. These are the general issues most in need of address in future development. Since the time (and memory) for singularity checks can often vastly outweigh the time for the main processing, for many of the intrinsics, singularity checks are *only* performed when the user explicitly asks for them by setting a parameter to `true`.

116.2 General Surfaces

116.2.1 Introduction

We describe here the newer functionality for surfaces (two-dimensional, geometrically integral schemes over a field) in arbitrary dimensional ambients. However, the reader should be aware that there are major restrictions for many of the intrinsics.

The biggest problems are the singularity assumptions (either non-singular or with at worst simple singularities) and restrictions to ordinary projective space which means that large dimensional ambients are unavoidable at times. These are the general issues most in need of address in future development. Since the time (and memory) for singularity checks can often vastly outweigh the time for the main processing, for many of the intrinsics, singularity checks are *only* performed when the user explicitly asks for them by setting a parameter value to `true`.

The main functionality is for the computation of fundamental invariants (irregularity, geometric genus etc.), checks for different type of 'non-singularity' (e.g. Gorenstein, only simple singularities), Kodaira-Enriques classification, computation of minimal models (including the full canonical model for a surface of general type) and construction of random surfaces from certain families in $\mathbf{P}^4$.

116.2.2 Creation Functions

As for general schemes and curves, surfaces may be created in any of Magma's ambient spaces. However, nearly all of the current specialised surface functionality only applies to surfaces in ordinary projective space.

The requirement for a scheme to be a surface is that it is defined over a field, of dimension 2, and is geometrically integral (reduced and irreducible when base extended to the algebraic closure of the ground field). Due to the difficulty in checking for *geometric* integrality, at present we only test for integrality (reduced and irreducible over the base field).

```
Surface(A,I)
Surface(A,f)
Surface(A,S)
```

`Nonsingular`	BoolElt	*Default :* `false`
`Check`	BoolElt	*Default :* `true`
`Saturated`	BoolElt	*Default :* `false`

Let A be an ambient or a scheme which already has some defining equations. The function returns the surface defined by the ideal I, the single polynomial f or the sequence of polynomials S within the scheme A.

If I or the set of new defining equations added to those of A generate an ideal that is known to be saturated (*c.f.* Section 112.3), `Saturated` can be set to `true`. If the surface is known to be non-singular or singular, much subsequent calculation can be avoided by setting `Nonsingular` to `true` or `false`. The parameter `Check` is `true` by default and forces the function to check that the surface is integral by testing primality of the (saturated) defining ideal. This can be an expensive computation in high-dimensional ambients, so it is best to set `Check` to `false` if it is known in advance that the surface is integral. As noted earlier, our actual requirement is that surfaces are geometrically irreducible (equivalently, the base field is integrally closed in the coordinate ring), because many of the surface intrinsics only really make sense for such varieties. However, this is a more difficult property to test. In practice, integrality should usually imply geometric integrality.

116.2.3 Invariants

The following functions give standard invariants for projective surfaces with only A-D-E singularities (or slightly weaker assumptions). Due to the current limitations of the cohomology and sheaf machinery, most are only available for ordinary projective surfaces. For corresponding functions that give invariants of the desingularization of hypersurfaces with more general singularities, see Section 116.3.4. Key invariants are stored when computed.

```
GeometricGenus(S)
```

`CheckGor` BoolElt *Default :* `false`

`UseCohom` BoolElt *Default :* `false`

The surface S should be an ordinary projective surface which is Gorenstein (this guarantees that a canonical sheaf K exists as a dualising sheaf and is invertible). Returns the geometric genus of S, defined as the dimension of the space of global sections of K, $h^0(K)$.

The boolean parameter `CheckGor` (default `false`) can be set to true to force a check that S is Gorenstein if this isn't already known and stored. By default, the computation computes (and stores) K and then does a direct computation of its global sections. The alternative method is to compute the genus via cohomology since the dimension of H^2 of the structure sheaf is equal to the genus. Set `UseCohom` to `true` to apply the second method. The advantage of the first method is that it is currently faster (in general) and also that K is used in many other intrinsics. Note that unless S is non-singular or has only A-D-E singularities, the genus computed here will generally be larger than the geometric genus of a desingularization of S.

```
Plurigenus(S,n)
```

`CheckGor` BoolElt *Default :* `false`

The surface S and parameter `CheckGor` are as in `GeometricGenus` above. The integer n should be non-negative. Returns the dimension of the space of global sections of the nth tensor power of the canonical sheaf K of S. Again this will generally be larger than the nth plurigenus of a desingularization of S unless S has at worst simple singularities.

`ArithmeticGenus(S)`

Given a scheme S, this function returns the arithmetic genus. It is, in fact, the general scheme intrinsic.

`Irregularity(S)`

CheckGor	BoolElt	*Default :* false
UseCohom	BoolElt	*Default :* false

The irregularity q of S, an ordinary projective surface, defined as the dimension of the cohomology group $H^1(S, O_S)$, where O_S is the structure sheaf of S.

If S is known to be Gorenstein or the geometric genus has already been computed and stored, this is computed from the geometric genus p_g and arithmetic genus p_a using the formula $q = p_g - p_a$. Note that S will be known Gorenstein if it is known to be non-singular or to only have simple singularities (All of these properties will have been stored if already tested for. See next section.).

If `CheckGor` is set to `true` (the default is `false`), and the above conditions arent satisfied, Gorensteinness will be checked and, if S is Gorenstein, the above procedure will be followed. Otherwise, the cohomology machinery is used directly. Setting `UseCohom` to `true` (the default is again `false`) will force the cohomology machinery to be used, unless the value of q has already been computed and stored.

`ChernNumber(S,n)`

CheckADE	BoolElt	*Default :* false

The surface S should be ordinary projective with at most simple (A-D-E) singularities. The integer n should be 1 or 2. The singularity condition, if not already known, will only be tested for if `CheckADE` is set to `true` (default is `false`). The function returns the nth Chern number of S_1, the *minimal* desingularization of S. For $n = 1$, this is just the intersection product $K.K$, where K is the canonical sheaf of S_1. Thanks to the singularity condition, this can just be computed on S. For $n = 2$, the Chern number $c_2(S)$ is computed from the relation $c_2(S) + K.K = 12 * (1 + p_a)$, where p_a is the arithmetic genus of S.

`MinimalChernNumber(S,n)`

CheckADE	BoolElt	*Default :* false

The surface S should be ordinary projective with at most simple (A-D-E) singularities. The integer n should be 1 or 2. The singularity condition, if not already known, will only be tested for if `CheckADE` is set to `true` (default is `false`). The function computes and returns the relevant Chern number for a minimal model S_2 of a desingularisation S_1 of S. As above, these numbers follow from knowing $K_m.K_m$ where K_m is the canonical sheaf of S_2. If k is the base-field and S is not rational or birationally ruled (i.e. of Kodaira dimension -1), then S_2 is defined over k and is unique up to k-isomorphism. In these cases, $K_m.K_m$ is known from the Kodaira dimension and the second plurigenus in the Kodaira dimension 2 (general type) case. For rational and ruled surfaces, the minimal model is not unique up to isomorphism and a geometrically minimal model may not be defined over k. In these cases, we conventionally take for the invariants a minimal model over the algebraic closure of k with maximal $K_m.K_m$, which is therefore 9 for rational S and 8 for non-rational, ruled S.

```
HodgeNumber(S,i,j)
```

`CheckADE` BoolElt *Default :* `false`

The surface S should be an ordinary projective with at most simple (A-D-E) singularities. The singularity condition, if not already known, will only be tested for if `CheckADE` is set to `true` (default is `false`).

The integers i, j should be such that $0 \leq i, j \leq 2$. The function returns the Hodge number $h^{i,j}$ of the minimal desingularization S_1 of S which is the dimension of the cohomology group $H^j(S_1, D^i)$ where D^i is the ith alternating power of the sheaf of differentials of S_1. These are computed by formula from the fundamental invariants which are the geometric genus, the irregularity and the first Chern number of S (or S_1).

Example H116E1

We take an easy example: the Kummer surface of the Jacobian of a genus 2 hyperelliptic curve, embedded in $\mathbf{P}^3$ as a degree 4 surface with 16 A_1 singularities lying beneath the 16 points of order 2 on the Jacobian. A nonsingular quartic in $\mathbf{P}^3$ is a K3 surface, but simple singularities don't affect the quartic being K3. We verify this here for the Kummer surface, finding that the invariants are the standard invariants for a K3 surface.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Surface(P,x^3*t+x^2*z^2-8*x*y^2*z-x*z*t^2+16*y^4+y^2*t^2-z^3*t);
> IsSingular(X);
true
> HasOnlySimpleSingularities(X);
true
> GeometricGenus(X);
1
> ArithmeticGenus(X);
1
> Irregularity(X);
0
> [ChernNumber(X,i) : i in [1,2]];
[ 0, 24 ]
> for i in [0..2], j in [0..2] do
>  printf "%o,%o : %o\n",i,j,HodgeNumber(X,i,j);
> end for;
0,0 : 1
0,1 : 0
0,2 : 1
1,0 : 0
1,1 : 20
```

```
1,2 : 0
2,0 : 1
2,1 : 0
2,2 : 1
```

116.2.4 Singularity Properties

This section contains intrinsics for testing for various levels of 'singularity' of a surface. There are further intrinsics applying to more general schemes in Chapter 112 for basic singularity/non-singularity as well as tests for whether a scheme is locally/arithmetically Cohen-Macaulay or locally/arithmetically Gorenstein. The tests here that are currently specific to surfaces are for normality and for having only simple (A-D-E) singularities. All of these properties are stored when computed for a surface/scheme and the various implications between them are used to shortcut tests. The intrinsics below rely on being able to compute the singular subscheme of the surface and having each singular point lying in a constructible affine patch, so they apply to surfaces lying in a wide range of ambients.

```
IsNormal(S)
```

Returns whether the surface S is a normal variety. The normality test used here consists of checking that the singular subscheme of S is empty or has dimension zero and applying a local normality test at each singular point p (over a splitting field). The local test used is simply whether the depth of the local ring is 2. Taking an affine patch at p and translating to the origin, we simply consider the quotient of the coordinate ring by a non-vanishing coordinate variable and check that the maximal homogeneous ideal is not an associated prime by a straightforward saturation computation. We could have also chosen to use our test for being Cohen-Macaulay once it is known that the singular subscheme of S is zero dimensional.

```
IsSimpleSurfaceSingularity(p)
```

The point p should be a point in the pointset of a surface S. It is referred to as a simple or A-D-E singularity if it is an isolated singularity on S which is analytically of the type A_n, $n \geq 1$, D_n, $n \geq 4$, E_6, E_7 or E_8 as described in Chapter III, Section 7 of [BHPdV04]. For convenience, if p is non-singular on S, we class it as a simple singularity of type A_0. These are all Gorenstein (even l.c.i) singularities. Their significance is that they are the surface singularities that impose no 'adjunction' condition on the canonical sheaf with respect to computing the canonical sheaf of the minimal desingularization S_1 of S. They all resolve to a collection of (-2)-curves on S_1 whose intersection pairing matrix is the negative of that of the root system with which they share a label.

This intrinsic tests whether p is a simple singularity and, if so, returns the type as a string ("A", "D" or "E") along with the index n (e.g. 6, 7 or 8 for type "E"). It requires that the characteristic of the base field of S is not 2. Also, the E_n types can be a little awkward to analyse in characteristic 3 so, in char 3, the intrinsic always returns `false` if p is a possible E type singularity.

The intrinsic first uses `IsHypersurfaceSingularity` to determine whether p is analytically isomorphic to a hypersurface singularity (which is the case for all simple singularities) and then tests for A-D-E type by examining the expansion of the equation that defines the analytically equivalent singularity.

NB: The intrinsic doesn't fully check that p is an isolated singularity (i.e., it doesn't lie on a curve in the singular locus of S). It may crash or hang in some cases where p is not isolated.

`HasOnlySimpleSingularities(S)`

`ReturnList` BoolElt *Default :* `false`

This intrinsic determines whether the surface S has no singularities worse than isolated simple singularities as described in the previous intrinsic. Again, the characteristic of the base field of S should not be 2. If S has only simple singularities and `ReturnList` is `true` (the default is `false`), a list is also returned containing triples that consist of each singular point of S (in a pointset over an extension of the base field) along with its type, given as a string and index number as described previously. In some cases, it may be already known (and internally stored) that there are only simple singularities. For example, if S is a minimal or weighted canonical model of a surface of general type.

Example H116E2

Anticanonically-embedded *degenerate* Del Pezzo surfaces of degree ≥ 3 are singular but have only simple singularities. We verify this for a degree 4 Del Pezzo which has 2 conjugate (over a quadratic extension) A_1 singularities.

```
> P<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> X := Surface(P,[x*z-y^2, t^2-2*u^2+x^2-2*z^2]);
> HasOnlySimpleSingularities(X : ReturnList := true);
true [* <(0 : 0 : 0 : r1 : 1), "A", 1>, <(0 : 0 : 0 : r2 : 1), "A", 1> *]
> _,lst := $1;
> Ring(Parent(lst[1][1]));
Algebraically closed field with 2 variables over Rational Field
Defining relations:
[
    r2^2 - 2,
    r1^2 - 2
]
```

116.2.5 Kodaira-Enriques Classification

```
KodairaEnriquesType(S)
```

`CheckADE` BoolElt *Default :* `false`

The argument S is a surface in ordinary projective space having at most simple (A-D-E) singularities. As it may be a very heavy computation, the latter is only checked if the user sets the `CheckADE` parameter is set to `true` (the default is `false`).

The function computes the type of S (or rather, of the non-singular projective surfaces in its birational equivalence class) according to the classification of Kodaira and Enriques.

The first number returned is the *Kodaira dimension* of S, which is -1, 0, 1, or 2. We use -1 here rather than $-\infty$. A second return value further specifies the type within the Kodaira dimension -1 or 0 cases (and is irrelevant in the other two cases).

Kodaira dimension -1 corresponds to birationally ruled surfaces. The second number returned in this case is the irregularity $q \geq 0$ of S. So S is birationally equivalent to a ruled surface over a smooth curve of genus q and is a *rational* surface if and only if q is zero.

Kodaira dimension 0 corresponds to surfaces which are birationally equivalent to a K3 surface, an Enriques surface, a torus or a bi-elliptic surface. In the final case, there is a partial subclassification in that the canonical sheaf of the minimal model is a torsion sheaf of order r, where r is 2, 3, 4, or 6. The second integer return value in the Kodaira dimension zero case codes the subtypes as follows:-

-3	Enriques surface
-2	K3 surface
-1	Torus
r	$r = 2, 3, 4$ or 6. Bielliptic surface of subtype r

A third return value is a string that gives a verbal description of the surface type (e.g. "Rational" or "Bi-elliptic (type 3)"). Kodaira dimension 1 surfaces are labelled as "Elliptic fibration" (which they all are – though there are also surfaces of Kodaira dimension less than 1 which have elliptic fibrations) and Kodaira dimension 2 surfaces are labelled as "General type", as is traditional.

There are no built-in restrictions on the characteristic of the base field, but there are some special cases for surfaces of Kodaira dimension 0 in characteristics 2 and 3 that may not be dealt with properly.

The function works by computing a number of the invariants of S and sometimes also considering the dimension of the image of appropriate pluri-canonical maps. We try to compute the least number of invariants to fully determine the type. So, a useful by-product is that, after calling this function, a number of the surface invariants (always including the geometric genus and irregularity) will have been computed and stored for later use. The type information is also stored.

```
KodairaEnriquesDimension(S)
```

CheckADE BoolElt *Default :* false

The argument S is a surface in ordinary projective space having at most simple (A-D-E) singularities. The later condition will only be checked if the parameter CheckADE is set to true. The function simply returns the Kodaira dimension (-1,0,1,2) without further type information. In most cases it does the same amount of work as to compute the full Kodaira-Enriques type, unless the result has already been determined and stored.

Example H116E3

We will present further cases of Kodaira-Enriques typing in our minimal model examples. For now, we just give two simple examples: a Veronese surface in $\mathbf{P}^5$ and the Kummer surface in $\mathbf{P}^3$ with simple singularities from our earlier example.

The scheme X is a Veronese surface, isomorphic to P^2:

```
> P<a,b,c,d,e,f> := ProjectiveSpace(Rationals(),5);
> X := Surface(P,[b^2-a*c, a*d-b*f, b*d-c*f, d^2-c*e, a*e-f^2, b*e-d*f]);
> // X is a Veronese surface, isomorphic to P^2
> KodairaEnriquesType(X);
-1 0 Rational
```

The scheme X is a singular K3 surface:

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Surface(P,x^3*t+x^2*z^2-8*x*y^2*z-x*z*t^2+16*y^4+y^2*t^2-z^3*t);
> KodairaEnriquesType(X);
0 -2 K3
```

116.2.6 Minimal Models

In contrast to the curve case, a birational equivalence class of surfaces contains an infinite number of non-isomorphic projective, non-singular surfaces. Any two such surfaces are linked by a birational map that consists of a sequence of blowing up points and blowing down exceptional curves (rational (-1)-curves). For any non-singular, projective surface, a sequence of blow downs of exceptional curves will result in a surface with no more exceptional curves after a finite number of steps. Such a surface is referred to as a *minimal model.* It is also possible to further contract connected cycles of rational (-2)-curves to simple singularities. Sometimes minimal model also refers to a surface on which these contractions have been performed. This is particularly true for surfaces of general type where the pluri-canonical models are minimal in this second sense.

For surfaces of Kodaira dimension greater than or equal to zero, there is a unique minimal model (up to isomorphism) within the birational equivalence class. That is, the minimisation procedure of blowing down exceptional curves will always lead to the same thing starting with any non-singular projective surface within that class. This can be carried out over an arbitrary base field and the minimal model is a unique representative of the class, which partly explains its importance.

For rational or ruled surfaces, there is not a unique minimal model. Over an algebraically closed field, the minimal models in these cases are the projective plane and the *geometrically ruled surfaces* which are fibrations over a non-singular, projective base curve C, all of whose fibres are irreducible curves isomorphic to the projective line. The rational surfaces are those with C rational (in this case, one of the ruled surfaces is not minimal but is the plane blown up in one point). Over a non-algebraically closed base field k, it may not be possible to blow down all exceptional curves working over k and so there are k-minimal surfaces (like certain Del Pezzo surfaces) that are not minimal over the algebraic closure. Minimal models are important for rational and ruled surfaces because they allow the reduction to a small class of standard isomorphism types

This section describes functions which are designed to construct minimal models or quasi-minimal models of ordinary projective surfaces. The precise meaning of this varies a little depending on the Kodaira dimension of S, so there are distinct functions for the different dimensions. The Kodaira dimension, if unknown, can be determined by use of the intrinsics in the previous section.

More of these intrinsics should work for surfaces with simple singularities - at least, if the user is happy with a result that also has simple singularities. For the moment, except for surfaces of general type (Kodaira dimension 2), we require S to be non-singular.

The output minimal models are all non-singular except for surfaces of general type where all (-2)-curves are contracted to simple singularities as is traditional. There is also an intrinsic to compute the full canonical model (which lies in weighted projective space in general) and the canonical coordinate ring of a surface of general type. With the Kodaira dimension 1 minimal models, the user can optionally ask for a map to a smooth projective curve that presents the minimal model M as an elliptic fibration. Thus, M is the global arithmetic minimal model of its generic fibre.

```
MinimalModelRationalSurface(S)
```

`CheckSing` BoolElt *Default :* `false`

Let S should be a non-singular ordinary projective rational (Kodaira dimension -1 and irregularity 0) surface. Non-singularity is not checked by default so to force a check, set `CheckSing` to `true`. It is also left to the user to check that S is rational (using Section KodairaEnriquesType for example).

The intrinsic does not strictly compute a minimal model M (there may be (-1)-curves that can still be blown down over the base field), but instead it produces a standard model that is terminal for the adjunction process.

This means that M will be a member of a small family of special rational surfaces: the projective plane or its Veronese embedding in $\mathbf{P}^5$, an anticanonically embedded Del Pezzo surface, a rational scroll or a conic bundle. Other functions for special surfaces may then be applied to M – to parametrize it, for example.

The return value is a rational map f from S onto M (M may be recovered as the codomain of f) which will be of type `MapSchGrph` or `MapSch`. The implementation proceeds by simply iterating the adjunction map until simple termination criteria are recognised.

Example H116E4

We take one of the family of non-singular rational surfaces in $\mathbf{P}^4$ that can be generated by the RandomRationalSurface_d10g9 intrinsic to be described later. These surfaces are very far from minimal. They have exceptional curves of degrees 3, 2 and 1 in the $\mathbf{P}^4$ embedding and the theory tells us that we need 2 adjunction maps to reduce the surface to a degree 5 Del Pezzo in $\mathbf{P}^5$, which is terminal. The degree of non-minimality is measured by the first Chern number $K.K$, with $9 - K.K$ or $8 - K.K$ telling us how many point blowups it takes to get from a geometric minimal model (over $\bar{k}$) to the surface, depending on whether the geometric minimal model is $\mathbf{P}^2$ or a rational ruled surface. Here, we start at -9 and get to 5 for the Del Pezzo, which is $\mathbf{P}^2$ blown up at 4 points over the algebraic closure of the base field. We choose to work over a finite field so that the example completes quickly.

```
> k := GF(37);
> P := ProjectiveSpace(k,4);
> X := RandomRationalSurface_d10g9(P);
> #DefiningPolynomials(X);
11
> [TotalDegree(f): f in DefiningPolynomials(X)];
[ 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
> // X is defined by a quartic and 10 quintics
> ChernNumber(X,1);
-9
> mp := MinimalModelRationalSurface(X);
> Y := Codomain(mp);
> Y;
Surface over GF(37) defined by
y[2]*y[3]+23*y[3]^2+4*y[1]*y[4]+29*y[2]*y[4]+34*y[3]*y[4]+36*y[4]^2+
 18*y[1]*y[5]+22*y[2]*y[5]+y[3]*y[5]+31*y[4]*y[5]+3*y[5]^2+33*y[1]*y[6]+
 6*y[2]*y[6]+31*y[3]*y[6]+20*y[4]*y[6]+28*y[5]*y[6]+33*y[6]^2,
y[1]*y[3]+27*y[3]^2+25*y[1]*y[4]+y[2]*y[4]+19*y[3]*y[4]+34*y[4]^2+13*y[1]*y[5]+
 15*y[2]*y[5]+14*y[3]*y[5]+15*y[4]*y[5]+17*y[5]^2+21*y[1]*y[6]+18*y[2]*y[6]+
 33*y[3]*y[6]+9*y[4]*y[6]+27*y[5]*y[6]+12*y[6]^2,
y[2]^2+13*y[1]*y[4]+10*y[2]*y[4]+9*y[3]*y[4]+26*y[4]^2+32*y[1]*y[5]+27*y[2]*y[5]+
 18*y[4]*y[5]+33*y[5]^2+27*y[1]*y[6]+16*y[2]*y[6]+31*y[3]*y[6]+35*y[4]*y[6]+
 24*y[5]*y[6]+21*y[6]^2,
y[1]*y[2]+26*y[3]^2+16*y[1]*y[4]+23*y[2]*y[4]+24*y[3]*y[4]+32*y[4]^2+27*y[1]*y[5]+
 14*y[2]*y[5]+10*y[3]*y[5]+12*y[4]*y[5]+33*y[5]^2+16*y[1]*y[6]+26*y[2]*y[6]+
 7*y[3]*y[6]+13*y[4]*y[6]+11*y[5]*y[6]+y[6]^2,
y[1]^2+5*y[3]^2+19*y[1]*y[4]+35*y[2]*y[4]+2*y[3]*y[4]+20*y[4]^2+29*y[1]*y[5]+
 27*y[2]*y[5]+22*y[3]*y[5]+14*y[4]*y[5]+6*y[5]^2+32*y[1]*y[6]+31*y[2]*y[6]+
 2*y[3]*y[6]+36*y[5]*y[6]+33*y[6]^2
> Ambient(Y); Degree(Y);
Projective Space of dimension 5 over Finite field of size 37
Variables: y[1], y[2], y[3], y[4], y[5], y[6]
```

```
5
> ChernNumber(Y,1);
5
```

```
MinimalModelRuledSurface(S)
```

CheckSing BoolElt *Default :* false

The surface S should be non-singular ordinary projective of Kodaira dimension -1. Non-singularity is not checked by default so to force a check, set CheckSing to true. It is also left to the user to check that S is of the correct Kodaira dimension (using KodairaEnriquesType for example).

If S is rational, the intrinsic MinimalModelRationalSurface is applied. For non-rational ruled surfaces, the same adjunction procedure is applied to lead to a terminal model M, which is either a non-rational scroll and genuinely minimal, a conic bundle over a non-rational curve that may not be strictly minimal (it may have degenerate fibres that are the intersecting unions of two (-1)-curves) or a non-split minimal ruled surface over a genus one curve embedded as a degree 9 surface in $\mathbf{P}^6$ in such a way that the fibres of the ruling have degree 3.

The return value is a rational map f from S onto M (M may be recovered as the codomain of f) and will be of type MapSchGrph or MapSch.

```
MinimalModelKodairaDimensionZero(S)
```

CheckSing BoolElt *Default :* false

The surface S should be non-singular ordinary projective of Kodaira dimension 0. Non-singularity is not checked by default so to force a check, set CheckSing to true. It is also left to the user to check that S is of the correct Kodaira dimension (using KodairaEnriquesType for example).

The function computes a minimal model M of S, again by repeatedly applying the adjunction map until minimality occurs (the first Chern number is zero).

The return value is a rational map f from S onto M (M may be recovered as the codomain of f) and will be of type MapSchGrph or MapSch.

Example H116E5

We start with an example of a non-minimal surface that is a torus T blown up in one point. Such examples naturally occur when T is the Jacobian J of a genus 2 curve C. The product of the C with itself quotiented by an appropriate involution is such a surface. They have an embedding in $\mathbf{P}^7$ which gives an image isomorphic to the projection from the zero point of J embedded in $\mathbf{P}^8$ by three times the theta divisor.

We work over a finite field for speed (although the example doesn't take too long over the rationals). Our example corresponds to the curve C with Weierstrass equation $y^2 = x^5 - 1$. The minimal model routine has the effect of *unprojecting* here: it recovers the torus J in its $\mathbf{P}^8$ embedding.

```
> P7<z1,z2,z3,z4,z5,z6,z7,z8> := ProjectiveSpace(GF(37),7);
> X := Surface(P7,
> [
> z3*z4-z2*z5+z1*z6,
> 1/2*z3^2-1/2*z2*z7+z1*z8,
> z1^2+z5^2-z4*z6-2*z3*z7+z2*z8,
> -z3*z5*z6+z2*z6^2+z5^2*z7-z4*z6*z7-z3*z7^2-1/2*z3^2*z8+1/2*z2*z7*z8,
> z3*z4*z6+1/2*z3^2*z7-z4*z5*z7+1/2*z2*z7^2+z2*z3*z8,
> -z4^2*z5+z2^2*z6+z6^3+z2*z4*z7+z6*z7*z8-z5*z8^2,
> -1/2*z3^2*z5-z4*z5^2+z2*z3*z6+z4^2*z6+z3*z4*z7+1/2*z2*z5*z7-z2*z4*z8,
> z1*z2*z5+z5^2*z6-z4*z6^2-z3*z6*z7-z5*z7^2+z3*z5*z8-z2*z6*z8+z4*z7*z8,
> -z4^3+z2^2*z5+z5*z6^2-z6*z7^2+2*z5*z7*z8-z4*z8^2,
> -z3*z4^2+z2^2*z7+z6^2*z7-z3*z8^2,
> -z2*z4^2+z1*z2*z7+z5*z6*z7+z3*z7*z8-z2*z8^2,
> -z2*z4^2+z1*z4*z5+z3*z6^2+z5^2*z8-z4*z6*z8,
> z1*z4^2-z3*z7^2-1/2*z3^2*z8+1/2*z2*z7*z8,
> 1/2*z3^2*z4-z2*z3*z5+z4^2*z5-z6^3-3/2*z2*z4*z7+z1*z5*z7-z6*z7*z8+z5*z8^2,
> -z2*z3*z4+z1*z4*z7+z6*z7^2+z3*z6*z8-z5*z7*z8,
> z2^2*z4-z5*z7^2-z2*z6*z8+z4*z7*z8,
> z1*z2*z4+1/2*z3^2*z6-z3*z5*z7+1/2*z2*z6*z7,
> -1/2*z3^3-z3*z4*z5+2*z2*z4*z6+3/2*z2*z3*z7-z4^2*z7+z1*z7^2,
> 1/2*z3^3+z3*z4*z5-z2*z4*z6-3/2*z2*z3*z7+z2^2*z8,
> -1/2*z2*z3^2-3/2*z3*z4^2+z2*z4*z5+z1*z3*z7+1/2*z6^2*z7-1/2*z3*z8^2,
> z2^2*z3-z2*z4^2+z3*z6^2+2*z3*z7*z8-z2*z8^2,
> z1*z2*z3+z3*z5*z6-z4*z6*z7-z3*z7^2+z3^2*z8+z2*z7*z8,
> z2^3+z3*z5*z6-z5^2*z7+1/2*z3^2*z8+1/2*z2*z7*z8,
> z1*z2^2-z3*z4*z6+z2*z5*z6-z3^2*z7
> ] : Check:= false);
> KodairaEnriquesType(X);
0 -1 Torus
> mp := MinimalModelKodairaDimensionZero(X);
> Y := Codomain(mp);
> Ambient(Y);
Projective Space of dimension 8 over Rational Field
Variables: y[1], y[2], y[3], y[4], y[5], y[6], y[7], y[8], y[9]
> Y;
Surface over Rational Field defined by
y[4]*y[5] - y[3]*y[6] + y[2]*y[7],
y[4]^2 - y[3]*y[8] + 2*y[2]*y[9],
y[3]*y[4] + y[5]^2 + y[1]*y[7] - 2*y[2]*y[8] + y[9]^2,
y[1]*y[4] + 2*y[2]*y[5] + y[7]*y[8],
y[3]^2 - 1/3*y[2]*y[4] + 1/3*y[1]*y[6] + 1/3*y[7]^2 + 1/3*y[8]*y[9],
y[2]*y[3] + y[1]*y[5] + y[6]*y[7] + y[8]^2 + 2*y[4]*y[9],
y[1]*y[3] - y[4]*y[7] + 2*y[6]*y[8],
y[2]^2 + y[6]^2 - y[5]*y[7] - 2*y[4]*y[8] + y[3]*y[9],
y[1]*y[2] + y[4]*y[6] - 2*y[3]*y[7] + y[5]*y[8],
y[3]*y[5]*y[7] - y[5]^2*y[8] + y[2]*y[8]^2 + 1/3*y[2]*y[4]*y[9] -
  1/3*y[1]*y[6]*y[9] - 1/3*y[7]^2*y[9] - 1/3*y[8]*y[9]^2,
y[2]*y[5]*y[7] + 2/3*y[2]*y[4]*y[8] + 1/3*y[1]*y[6]*y[8] +
  1/3*y[7]^2*y[8] - y[1]*y[5]*y[9] - y[6]*y[7]*y[9] - 2/3*y[8]^2*y[9] -
  2*y[4]*y[9]^2,
y[2]*y[5]*y[6] + y[4]*y[7]^2 + y[1]*y[5]*y[8] + y[8]^3 + y[6]^2*y[9] -
  y[5]*y[7]*y[9] + y[4]*y[8]*y[9] + y[3]*y[9]^2
> MinimalChernNumber(X,1)-ChernNumber(X,1);
1
> ChernNumber(Y,1);
0
```

So we have one exceptional line blown down and Y is now minimal.

Example H116E6

For our second example, we take a surface of Enriques type in $\mathbf{P}^4$ which comes from one of the special families of $\mathbf{P}^4$ surfaces to be described later. Such surfaces are non-minimal, but isomorphic to an Enriques surface blown up at one point. As we are choosing a random surface from a family which will be defined by many non-sparse equations, we work over a finite field for speed.

```
> k := GF(37);
> P := ProjectiveSpace(k,4);
> X := RandomEnriquesSurface_d9g6(P);
> #DefiningPolynomials(X);
15
> [TotalDegree(f): f in DefiningPolynomials(X)];
[ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
> //surface defined by 15 quintics
> ChernNumber(X,1);
-1
> mp := MinimalModelKodairaDimensionZero(X);
> Y := Codomain(mp);
> #DefiningPolynomials(Y);
10
> [TotalDegree(f): f in DefiningPolynomials(Y)];
[ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
> // minimal model Y defined by 10 cubics
> Ambient(Y); Degree(Y);
Projective Space of dimension 5 over Finite field of size 37
Variables: y[1], y[2], y[3], y[4], y[5], y[6]
10
> ChernNumber(Y,1);
0
```

```
MinimalModelKodairaDimensionOne(S)
```

CheckSing	BoolElt	*Default :* false
Fibration	BoolElt	*Default :* false

Let S be a non-singular ordinary projective surface of Kodaira dimension 1. Non-singularity is not checked by default and to force a check, set `CheckSing` to `true`. It is also left to the user to check that S is of the correct Kodaira dimension (using `KodairaEnriquesType` for example).

The function computes a minimal model M of S. Such a model always has a connected elliptic fibration to a smooth projective curve C (i.e. there is a morphism g from M onto C such that the general fibre of g is a smooth curve of genus 1). If `Fibration` is set to `true` (the default is `false`), the function also computes g and returns it as a second return value.

The first return value is a rational map f from S onto M (M may be recovered as the codomain of f) and will be of type `MapSchGrph` or `MapSch`.

The computation of f again proceeds by repeating adjunction until the first Chern number is zero. There is a slight speed-up here, though, that we apply when S has positive geometric genus. We use the appropriate modification of the image of an effective canonical divisor to compute a new canonical divisor at each stage of adjunction. Although this doesn't usually speed up the computation of the next adjunction map, it can greatly increase the speed of computation of the new first Chern number, which speeds up testing of the termination criterion.

If required, g may be computed by using an appropriate small multiple of the pluricanonical map on M.

```
MinimalModelGeneralType(S)
```

```
CanonicalWeightedModel(S)
```

CheckADE	BoolElt	*Default :* false

The surface S should be ordinary projective of general type (Kodaira dimension 2) with at worst simple (A-D-E) singularities. The singularity condition is not checked by default and to force a check, set `CheckADE` to `true`. It is also left to the user to check that S is of general type (using `KodairaEnriquesType` for example).

The intrinsics construct a minimal model M for S which is of the canonical type: it has any (-2)-curves contracted to simple singularities.

The first intrinsic produces M as an ordinary projective surface as before, which is an m-canonical embedding with m equal to 3 generally but 4 or 5 when S has certain small invariants. This is implemented simply by computing the m-canonical map on S (this map automatically factors through a non-singular minimal model). The drawback of this is that M is usually defined in a high-dimensional projective space. There is a very simple check to see whether the canonical sheaf or the bi-canonical sheaf is very ample, in which case M is just taken as S.

The second intrinsic actually computes the *full* canonical model of S. This is a (generally weighted) projective model of M equal to *Proj* of the canonical coordinate ring $\sum_{n=0}^{\infty} H^0(S, K_S^{\otimes n})$, where K_S is the canonical sheaf of K. This is more intrinsic and it generally gives a model in a much lower-dimensional space with the drawback that this space is now weighted. The cases where the canonical or bicanonical image is actually isomorphic to M will show up in this version whereas it will produce an actual ordinary projective model in the first case. Currently, this intrinsic has the small restriction that S cannot have geometric genus (and irregularity) zero. The implementation involves calculating with Riemann-Roch spaces for small multiples of an effective canonical divisor.

For either intrinsic, the first return value is a rational map f from S onto M (M may be recovered as the codomain of f) which will be of type `MapSchGrph` or `MapSch` for the first intrinsic and of type `MapSch` only for the second. A second return value is a boolean with value `true` if and only if the original model X is minimal in the sense of containing no (-1)-curves.

```
CanonicalCoordinateIdeal(S)
```

`CheckADE` BoolElt *Default :* `false`

The surface S should be ordinary projective of general type (Kodaira dimension 2) with at worst simple (A-D-E) singularities. The singularity condition is not checked by default and to force a check, set `CheckADE` to `true`. It is also left to the user to check that S is of general type (using `KodairaEnriquesType` for example).

This intrinsic constructs the full canonical coordinate ring of S as described for `CanonicalWeightedModel`. What is returned is a homogeneous ideal I in a polynomial ring R with variable weightings over the base field of S such that the canonical coordinate ring is isomorphic to the quotient R/I as a graded ring.

Example H116E7

We give two examples of computing minimal models for some surfaces of general type. We begin with a non-singular type I Horikawa surface with first Chern number K^2 equal to 3. It is already minimal and the bi-canonical map gives an embedding into $\mathbf{P}^6$. We start with a bicanonical model. The first intrinsic luckily recognises that the surface is bi-canonical and just returns it as a minimal model. The second intrinsic finds the well-known weighted-projective embedding for such surfaces as a sextic hypersurface in a $\mathbf{P}(1,1,1,2)$ weighted projective space. (see Section 9, Chapter VII, [BHPdV04] or [Hor76]).

```
> P<x1,x2,x3,x4,x5,x6,x7> := ProjectiveSpace(Rationals(),6);
> X := Surface(P,[
>  -x4*x5 + x2*x6, x1*x5 - x4*x6,
>   x3*x4 - x5*x6, x2*x3 - x5^2,
>   x1*x3 - x6^2, x1*x2 - x4^2,
>   x1^3+x1*x2^2+x2^2*x3-x2*x3^2+x3^3+x2^2*x4+x1*x3*x4-x2*x3*x4+x1*x4^2-
>    x2*x4^2-x3*x4^2-x1*x2*x5-x1*x3*x5+x2*x3*x5+x1*x4*x5-x2*x4*x5+x4^2*x5-
>    x1*x5^2+x2*x5^2-x3*x5^2-x5^3-x1^2*x6+x1*x2*x6-x2*x3*x6-x2*x4*x6+
>    x1*x5*x6-x3*x5*x6-x4*x5*x6+x5^2*x6+x1*x6^2+x2*x6^2-x3*x6^2+x5*x6^2-
>    x1*x2*x7+x1*x3*x7+x2*x3*x7-x3^2*x7+x1*x4*x7-x2*x4*x7-x4^2*x7+x5^2*x7+
>    x2*x6*x7+x3*x6*x7-x4*x6*x7+x5*x6*x7-x6^2*x7+x3*x7^2+x4*x7^2+x5*x7^2-
>    x6*x7^2-x7^3 ]);
> IsSingular(X);
false
> KodairaEnriquesType(X);
2 0 General type
> mp := MinimalModelGeneralType(X);
> X1 := Codomain(mp); //the minimal model
> X1 eq X;
true
> mp1 := CanonicalWeightedModel(X);
> Y := Codomain(mp1);
> PW<a,b,c,d> := Ambient(Y);
> PW;
Projective Space of dimension 3 over Rational Field
Variables: a, b, c, d
The grading is:
    1, 1, 1, 2
```

The fact that the full weighted canonical model has only weights 1 and 2 also shows that the bi-canonical map is an embedding.

```
> Y;
Surface over Rational Field defined by
a^6 + a^4*b^2 + a*b^5 - a^5*c + 2*a^3*b^2*c - a^2*b^3*c - a*b^4*c + a^4*c^2 +
 2*a^3*b*c^2 - 2*a^2*b^2*c^2 - a*b^3*c^2 + 2*b^4*c^2 - a^2*c^4 - a*b*c^4 -
 2*b^2*c^4 + c^6 + a^3*b*d - 2*a^2*b^2*d - a*b^3*d - a^2*b*c*d + a*b^2*c*d +
 a*b*c^2*d + 2*b^2*c^2*d + a*c^3*d - c^4*d + a*b*d^2 - a*c*d^2 + b*c*d^2 +
 c^2*d^2 - d^3
```

We can also ask for the canonical coordinate ideal that defines the canonical coordinate ring which is the coordinate ring of Y. The call takes no time as the canonical weighted model Y and map `mp1` from X to Y have been stored internally.

```
> time CanonicalCoordinateIdeal(X);
Ideal of Graded Polynomial ring of rank 4 over Rational Field
Order: Grevlex with weights [1, 1, 1, 2]
Variables: a, b, c, d
Variable weights: [1, 1, 1, 2]
Homogeneous, Dimension >0
Groebner basis:
[
a^6 + a^4*b^2 + a*b^5 - a^5*c + 2*a^3*b^2*c - a^2*b^3*c - a*b^4*c + a^4*c^2 +
 2*a^3*b*c^2 - 2*a^2*b^2*c^2 - a*b^3*c^2 + 2*b^4*c^2 - a^2*c^4 - a*b*c^4 -
 2*b^2*c^4 + c^6 + a^3*b*d - 2*a^2*b^2*d - a*b^3*d - a^2*b*c*d + a*b^2*c*d +
 a*b*c^2*d + 2*b^2*c^2*d + a*c^3*d - c^4*d + a*b*d^2 - a*c*d^2 + b*c*d^2 +
 c^2*d^2 - d^3
]
Time: 0.000
```

Example H116E8

This example is a non-singular but non-minimal surface. It is a surface in $\mathbf{P}^8$ which is isomorphic to the blow-up of a point on a degree 5 hypersurface. The weighted canonical model is just the hypersurface in ordinary projective 3-space with the exceptional line blown back down. The computation of the canonical model takes a little time.

```
> P<x1,x2,x3,x4,x5,x6,x7,x8,x9> := ProjectiveSpace(Rationals(),8);
> X := Surface(P,[
>  x5*x6 - x7*x9, -x5*x8 + x4*x9, x4*x6 - x7*x8, -x5*x7 + x3*x9,
> -x4*x7 + x3*x8, x3*x6 - x7^2, x2*x7 - x5*x8 - x5*x9,
> x2*x6 - x8*x9 - x9^2, x2*x3 - x4*x5 - x5^2, -x2*x8 + x1*x9,
> x1*x7 - x4*x8 - x5*x8, x1*x6 - x8^2 - x8*x9, -x2*x4 + x1*x5,
> x1*x3-x4^2-x4*x5,
> x6^4+x3*x7^3+x1*x8^3-x2*x8^3+x2*x8^2*x9-x2*x8*x9^2+x2*x9^3,
> x6^3*x7+x3^2*x7^2+x1*x4*x8^2-x2*x4*x8^2+x2*x5*x8^2-x2*x5*x8*x9+x2*x5*x9^2,
> x3^3*x7+x6^2*x7^2+x1*x4^2*x8-x2*x4^2*x8+x2*x4*x5*x8-x2*x5^2*x8+x2*x5^2*x9,
> x3^4+x1*x4^3-x2*x4^3+x2*x4^2*x5-x2*x4*x5^2+x2*x5^3+x6*x7^3,
> x3*x4*x7^2+x3*x5*x7^2+x6^3*x8+x1^2*x8^2-x1*x2*x8^2+x2^2*x8^2+x6^3*x9-
>  x2^2*x8*x9+x2^2*x9^2,
> x3^2*x4*x7+x3^2*x5*x7+x1^2*x4*x8-x1*x2*x4*x8+x2^2*x4*x8-x2^2*x5*x8+
>  x6^2*x7*x8+x2^2*x5*x9+x6^2*x7*x9,
> x3^3*x4+x1^2*x4^2-x1*x2*x4^2+x2^2*x4^2+x3^3*x5-x2^2*x4*x5+x2^2*x5^2+
>  x6*x7^2*x8+x6*x7^2*x9,
> x3*x4^2*x7+2*x3*x4*x5*x7+x3*x5^2*x7+x1^3*x8-x1^2*x2*x8+x1*x2^2*x8-x2^3*x8+
>  x6^2*x8^2+x2^3*x9+2*x6^2*x8*x9+x6^2*x9^2,
> x1^3*x4-x1^2*x2*x4+x1*x2^2*x4-x2^3*x4+x3^2*x4^2+x2^3*x5+2*x3^2*x4*x5+x3^2*x5^2+
>  x6*x7*x8^2+2*x6*x7*x8*x9+x6*x7*x9^2,
> x1^4-x1^3*x2+x1^2*x2^2-x1*x2^3+x2^4+x3*x4^3+3*x3*x4^2*x5+3*x3*x4*x5^2+x3*x5^3+
>  x6*x8^3+3*x6*x8^2*x9+3*x6*x8*x9^2+x6*x9^3] : Check := false,
>    Nonsingular := true);
> KodairaEnriquesType(X);
2 0 General type
> mp, is_min := CanonicalWeightedModel(X);
> is_min;
false
> Codomain(mp); //the canonical model
Surface over Field defined by
$.1^5 + $.2^5 + $.3^5 + $.4^5
> MinimalChernNumber(X,1) - ChernNumber(X,1);
1
```

So there is one exceptional divisor blown down.

116.2.7 Special Surfaces in Projective 4-space

There has been much study of the families of nonsingular surfaces of non-general type that lie in $\mathbf{P}^4$ (a complete intersection of two nonsingular hypersurfaces of degrees d and e – the obvious construction – leads to general-type surfaces unless $d + e \leq 5$). These families are a very useful tool for generating random surfaces of a particular Kodaira-Enriques type with a non-singular ordinary projective model lying in a small-dimensional ambient.

Decker, Ein and Schreyer have given many such families in their paper [DES93]. Their constructions make use of the theory of codimension two schemes and the Beilinson spectral sequence for coherent sheaves. The result is that the defining ideal of a surface in the family is isomorphic as a graded module to the cokernel of a random map between two modules over $k[x_0, \ldots, x_4]$ that are direct sums of twists of the modules representing alternating powers of the sheaf of differentials on $\mathbf{P}^4$ (or something slightly more complex derived from these modules). The ideals constructed give Cohen-Macaulay surfaces and the surface is actually non-singular for a general map. The families are generally described by the degree, sectional genus (arithmetic genus of a hyperplane section) and Kodaira-Enriques type of the surfaces they contain. Given the type, knowing the degree and sectional genus is equivalent to knowing the Hilbert polynomial.

Intrinsics are provided to generate a random surface from a selection of the families described in the paper above, following the implementations in the Macaulay 2 computer algebra package.

```
RandomRationalSurface_d10g9(P)
```

RndP	RngIntElt	*Default* : 2
Check	BoolElt	*Default* : true

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. The intrinsic generates a random (non-minimal) degree 10 rational surface with sectional genus 9 in P from the family described by Ranestad. If the rationals are the base field, the parameter `RndP` is a positive integer used as an upper absolute bound for random coefficients of polynomials. That is, the algorithm uses random integers between $-RndP$ and $+RndP$m, inclusive. The default value here is 2. Parameter `Check` being `true` (the default) means that the random prospective surfaces generated are tested for irreducibility and non-singularity and rejected if they fail (the process tries 10 surfaces before giving up). The user can set this to `false` for a slight speed-up: it is rare that a singular surface occurs.

```
RandomEnriquesSurface_d9g6(P)
```

RndP	RngIntElt	*Default* : 2
Check	BoolElt	*Default* : true

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random (non-minimal) degree 9 Enriques surface with sectional genus 6 in P. The parameters have the same meaning as for `RandomRationalSurface_d10g9` and the same defaults.

```
RandomAbelianSurface_d10g6(P)
```

`RndP`	RngIntElt	*Default :* 5
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random degree 10 torus with sectional genus 6. This is just the zero section of an element of the Horrocks-Mumford bundle. The parameters have the same meaning as for `RandomRationalSurface_d10g9` although the default for `RndP` is now 5.

```
RandomEllipticFibration_d7g6(P)
```

`RndP`	RngIntElt	*Default :* 2
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random (minimal) degree 7 elliptic surface with sectional genus 6 in P. These have Kodaira dimension 1, geometric genus 2 and irregularity 0. The parameters have the same meaning as for `RandomRationalSurface_d10g9` and the same defaults.

```
RandomEllipticFibration_d8g7(P)
```

`RndP`	RngIntElt	*Default :* 2
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random (minimal) degree 8 elliptic surface with sectional genus 7 in P. These have Kodaira dimension 1, geometric genus 2 and irregularity 0. The parameters have the same meaning as for `RandomRationalSurface_d10g9` and the same defaults.

```
RandomEllipticFibration_d9g7(P)
```

`RndP`	RngIntElt	*Default :* 2
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random (minimal) degree 9 elliptic surface with sectional genus 7 in P. These have Kodaira dimension 1, geometric genus 1 and irregularity 0. The parameters have the same meaning as for `RandomRationalSurface_d10g9` and the same defaults.

```
RandomEllipticFibration_d10g10(P)
```

RndP	RngIntElt	*Default :* 2
Check	BoolElt	*Default :* true

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random (non-minimal) degree 10 elliptic surface with sectional genus 10 in P. These have Kodaira dimension 1, geometric genus 2 and irregularity 0. The parameters have the same meaning as for `RandomRationalSurface_d10g9` and the same defaults.

116.3 Surfaces in $\mathbf{P}^3$

116.3.1 Introduction

This section describes several packages of functionality developed for working with (hyper)surfaces in three-dimensional projective space $\mathbf{P}^3$.

At the core is a package to compute a **formal desingularization** of such a hypersurface X, expressed via a collection of algebraic power series giving the formal completion of the components of *some* desingularization lying over the components of the singular subscheme of the hypersurface. This allows the computation of important birational invariants of *any* desingularization of X like the arithmetic and geometric genera and higher geometric plurigenera. The algorithm is based on the method of Jung and was designed and implemented by Tobias Beck. It is fully described in [Bec07].

An important application of the desingularization data is the computation of *m-adjoint* maps as rational maps on X. A function is provided for this. Theoretical and algorithmic details may be found in [BS08].

There are functions to determine whether X is of Kodaira dimension $-\infty$, *i.e.,* birationally ruled. For the important special case of *rational* surfaces, there is a suite of functions to determine whether a parameterization exists over the base field and to explicitly construct one in the affirmative case. This is based on the work of Josef Schicho described in [Sch98] and [Sch00].

A surface X is mapped to a standard model by applying an appropriate m-adjoint map. These are then parameterized by special case code. The main special cases are Del Pezzo surfaces (including some singular cases) and line and conic bundles. The functions can be called directly by the user. Apart from the previously existing Del Pezzo code (for degrees 6, 8 and 9) and the special singular code for degrees 3 and 4, these functions were implemented by Tobias Beck and Josef Schicho.

116.3.2 Embedded Formal Desingularization of Curves

A formal embedded desingularization of plane curves, as described below, is used in the Jung surface resolution process. The main function is available to the user and provides another alternative to the existing function field and resolution graph curve functionality.

Before describing the function, we introduce some terminology. Let $C \subset P$ be a plane algebraic curve (where $P = \mathbf{A}^2_{\mathbf{E}}$ or $P = \mathbf{P}^2_{\mathbf{E}}$ for some field $\mathbf{E}$ of characteristic zero) and

$\pi : Q \to P$ an *embedded desingularization*, *i.e.*, π is proper birational, Q is regular and $D := \pi^{-1}(C) \subset Q$ is a normal crossing divisor. Further let $\{p_1, \dots, p_r\} \in Q$ be the generic points of the decomposition of D into irreducible components and $\{q_1, \dots, q_s\} \in Q$ the closed points of the normal crossings of D.

From π we can construct morphisms Spec $\widehat{\mathcal{O}_{Q,p_i}} \to P$ and Spec $\widehat{\mathcal{O}_{Q,q_i}} \to P$. The set of all these morphisms (up to isomorphism of the domain) is called a *formal embedded desingularization* of $C \subset P$. Each of these morphisms has a centre on P which is defined to be the image of the closed point.

The two classes of morphisms are represented, respectively, by homomorphisms $A \to \widehat{\mathcal{O}_{Q,p_i}}$ and $A \to \widehat{\mathcal{O}_{Q,q_i}}$ (where A is either the normal polynomial ring $\mathbf{E}[x, y]$ or the graded polynomial ring $\mathbf{E}[x, y, w]$ and the inverse image of the maximal ideal of the completion ring is the prime ideal defining the centre), and we are free to choose an isomorphic representation of the codomain. We refer to the homomorphisms as μ_i and ν_i respectively.

```
ResolveAffineCurve(p)
```

`Factors`	SeqEnum	*Default :* []
`Ps`	RngMPolElt	*Default :* 0
`Focus`	RngMPolElt	*Default :* 0
`ExtName`	MonStgElt	*Default :* "*alpha*"
`ExtCount`	RngIntElt	*Default :* 0
Verbose	`Resolve`	*Maximum :* 1

Given the curve defined by $p \in \mathbf{E}[x, y]$ (a non-zero bivariate polynomial over a number field), this intrinsic essentially computes a formal embedded resolution of the curve using a succession of point blow ups. Only morphisms whose centres vanish on the ideal generated by `Focus` are considered. Note that `Focus` may be a single polynomial or a sequence of polynomials.

The three returned lists contain elements of the form $(b_1, (y, m_{11}), (p_1, m_{12}))$, $(b_2, (y, m_2))$ and $(b_2, (p_3, m_3))$ respectively. Here b_1, b_2 and b_3 are homomorphisms $\mathbf{E}[x, y] \to \mathbf{E}'[x, y]$ to some bivariate polynomial ring over an algebraic field extension $\mathbf{E}'$ over $\mathbf{E}$.

The first list gathers normal crossings. Precisely, the extended homomorphism $b_1 : \mathbf{E}[x, y] \to \mathbf{E}'[[x, y]]$ corresponds to a ν_i from above. Moreover we have $\langle b_1(p)\rangle = \langle y^{m_{11}} p_1^{m_{12}}\rangle$ where $y = 0$ and $p_1 = 0$ have a normal crossing.

The second list gathers exceptional divisors. The extended homomorphism $b_2 : \mathbf{E}[x, y] \to \mathbf{E}'(x)[[y]]$ corresponds to a μ_i from above. Moreover we have $\langle b_2(p)\rangle = \langle y^{m_2}\rangle$ and $y = 0$ corresponds to an exceptional divisor.

Finally, the last list corresponds to the components of the original curve. The extended homomorphism $b_3 : \mathbf{E}[x, y] \to \mathbf{E}'\widehat{[x, y]}_{\langle p_3\rangle}$ (where $\mathbf{E}' = \mathbf{E}$ in this case) corresponds to another μ_i from above. Moreover we have that $b_3(p)$ has multiplicity m_3 in $\mathbf{E}'\widehat{[x, y]}_{\langle p_3\rangle}$ and corresponds to an original curve component.

If known, a factorization of p (as returned by the `Factorization` command) can be passed using the parameter `Factors` and the squarefree part of p (as returned by the `SquarefreePart` command) using `Ps`.

If the ground field has to be extended, the algebraic elements will be displayed as ExtName_i where i starts from ExtCount. The last return value is the value of ExtCount plus the number of field extensions that have been introduced, which can be useful for consecutive naming when making a series of resolution calls.

Example H116E9

We compute an embedded resolution of an affine plane curve.

```
> Q := Rationals();
> Qxy<x,y> := PolynomialRing(Q, 2, "glex");
> f := (y^2-x^3)*(x^2-y^2-y^3);
> NCs, EXs, DCs := ResolveAffineCurve(f : Factors := Factorization(f));
> #NCs, #EXs, #DCs;
7 4 2
> NCs[2]; EXs[3]; DCs[1];
[*
    Mapping from: RngMPol: Qxy to RngMPol: Qxy,
    <y, 4>,
    <-x^2 + 2*x + y, 1>
*]
[*
    Mapping from: RngMPol: Qxy to RngMPol: Qxy,
    <y, 10>
*]
[*
    Mapping from: RngMPol: Qxy to RngMPol: Qxy,
    <y^3 - x^2 + y^2, 1>
*]
> NCs[2][1](x), NCs[2][1](y);
x*y - y
y
```

Here we have passed the factorization of f only for illustrative purposes. The curve is the union of a cusp and a node at the origin. It has two singular points over $\mathbf{Q}$, the origin and another intersection point of the two curves which has a residue field of degree 5 over $\mathbf{Q}$.

We have computed the local information of a (not necessarily minimal) embedded resolution and find that it contains the 2 components of the strict transform, further 4 exceptional divisors and 7 normal crossings. For example, the pushforward of f under the chart map $x \mapsto xy - y, y \mapsto y$ is equal to $y^4(-x^2 + 2x + y)$ up to a local unit. The corresponding germ is isomorphic to a normal crossing in the embedded desingularization. We also see that one of the exceptional divisors has multiplicity 10.

If we were only interested in a local resolution, we would do the following:

```
> NCs, EXs, DCs := ResolveAffineCurve(f : Focus := [x,y]);
> #NCs, #EXs, #DCs;
5 3 0
```

We focus on the origin, hence, any curve components are not considered . We have 1 less exceptional divisor and 2 less normal crossings. This is because the second intersection point of the above two curve components was already a normal crossing, but our algorithm has nevertheless blown it up in the previous example.

`ResolveProjectiveCurve(p)`

`Focus`	RngMPolElt	*Default :* 0
`ExtName`	MonStgElt	*Default :* "*alpha*"
`ExtCount`	RngIntElt	*Default :* 0
Verbose	`Resolve`	*Maximum :* 1

Given the curve defined by $p \in \mathbf{E}[x, y, z]$ (a non-zero trivariate homogeneous polynomial over a number field), this intrinsic essentially computes a formal embedded resolution of the curve using a succession of point blow ups. This is the same as `ResolveAffineCurve` above, but now p is a homogeneous polynomial in three variables that defines a projective curve. Accordingly, the b_j map from the respective homogeneous coordinate ring to some $\mathbf{E}'[x, y]$.

Example H116E10

We can also desingularize the projectivisation of the above curve.

```
> Q := RationalField();
> QXYZ<X,Y,Z> := PolynomialRing(Q, 3);
> F := (Y^2*Z-X^3)*(X^2*Z-Y^2*Z-Y^3);
> NCs, EXs, DCs := ResolveProjectiveCurve(F); #NCs, #EXs, #DCs;
7 4 2
> NCs[3];
[*
    Mapping from: RngMPol: QXYZ to Polynomial ring of rank 2 over
    Rational Field ...,
    <y, 4>,
    <x^2 + 2*x - y, 1>
*]
> NCs[3][1](X);
x*y + y
> NCs[3][1](Y);
y
> NCs[3][1](Z);
1
```

The homomorphisms take a slightly different shape (because they have now $\mathbf{Q}[X, Y, Z]$ as domain), but otherwise they are the same. This is because the curve has no singularities at infinity.

116.3.3 Formal Desingularization of Surfaces

For curves we have described embedded formal desingularization. For surfaces instead we produce only *formal desingularizations*. Let $S \subset P$ be a hypersurface (where $P = \mathbf{A}^3_{\mathbf{E}}$ or $P = \mathbf{P}^3_{\mathbf{E}}$) and $C \subset S$ a closed subset (which typically contains the singular locus). Further let $\pi : T \to S$ be a *desingularization*, *i.e.*, π is proper birational and T is regular. By $\{p_1, \ldots, p_r\} \in T$ we denote the generic points of the curve components of the decomposition of $D := \pi^{-1}(C)$ into irreducibles.

From π we can construct morphisms Spec $\widehat{\mathcal{O}_{T,p_i}} \to S$. The set of all these morphisms (up to isomorphism of the domain) is called a *formal desingularization* of S over $C \subset S$. Such a morphism has a centre on S which is defined as the image of the closed point (and actually is contained in C).

The morphisms are represented by homomorphisms $A \to \widehat{\mathcal{O}_{T,p_i}}$ (where A is either the algebra $\mathbf{E}[x, y, z]/\langle p\rangle$ or the graded algebra $\mathbf{E}[x, y, z, w]/\langle p\rangle$ with p a defining polynomial), and we are free to choose an isomorphic representation of the codomain. We refer to such a homomorphisms as μ_i.

In the actual algorithm, C is the ramification locus of a finite projection, *pr*, to an affine or projective plane (C contains the singular subscheme of S). The underlying desingularization T (which is not computed explicitly) is a Jung resolution which is constructed in two stages. Firstly, an embedded resolution of the image of C in the plane is performed by blow-ups and T_1 is taken as the normalization of the pullback of this by *pr*. So T_1 then has only point singularities of a simple type (*toric singularities*), lying over the (normal-crossing) intersections of components of the embedded resolution. These are resolved by a finite succession of blow-ups on T_1 to give T.

The algorithm computes the formal desingularization, as described above, corresponding to T, using the embedded formal desingularization for curves followed by algebraic power series operations for the normalization and final resolution of the toric singularities. This is described fully in [Bec07]. The μ_i homomorphisms are defined by algebraic power series images of the variables of P.

It is important to note that the Jung desingularization T is not a minimal desingularization and, in any case, the set of morphisms returned for the formal desingularization generally contain some elements whose centre on S is already non-singular (because, for example, components of C are often generically non-singular). However, there is an parameter option with the main function `ResolveProjectiveSurface`, which removes "non-singular" morphisms and possibly others that have no effect on the computation of birational invariants and m-adjoint maps.

```
ResolveAffineMonicSurface(s)
```

`Focus`	RngMPolElt	*Default :* 0
`ExtName`	MonStgElt	*Default :* "*alpha*"
`ExtCount`	RngIntElt	*Default :* 0
Verbose	`Resolve`	*Maximum :* 1

The main user resolution function `ResolveProjectiveSurface` is for projective hypersurfaces. This affine version, however, may be useful in some circumstances.

The input is a monic, squarefree polynomial $s \in \mathbf{E}[x, y][z]$ where $\mathbf{E}$ is a number field (*i.e.*, s is univariate over a bivariate polynomial ring). Let $S \subset \mathbf{A}^3_{\mathbf{E}}$ denote the surface defined by it and $C \subset S$ the closed subset defined by $\mathrm{disc}_z(s)$ (*i.e.*, the intersection of S with the cylinder over the discriminant curve when considering the projection $S \to \mathbf{A}^2_{\mathbf{E}}$ in z-direction). The function computes a formal desingularization of S over C (see above).

The first return value is a list of elements of the form $((X, Y, Z), o)$ where $X, Y, Z \in \mathbf{F}[[t]]$ are univariate power series (over some field extension $\mathbf{F}$ of transcendence degree 1 over $\mathbf{E}$) s.t. $s(X, Y, Z) = 0$ and o is an integer. The induced homomorphism $\mathbf{E}[x, y][z]/(s) \to \mathbf{F}[[t]]$ corresponds to a μ_i from above and o is its *adjoint order*, *i.e.*, the negation of the order of a special differential form (see Section 116.3.4).

One can specify a focus ideal $F \subset \mathbf{E}[x, y]$ by passing a single generator or sequence of generators in `Focus` (as for `ResolveAffineCurve`). In this case C is taken to be the intersection of S and the cylinder over the zero set of $F + \langle \mathrm{disc}_z(s) \rangle$.

If the ground field has to be extended, the algebraic elements will be displayed as `ExtName_i` where `i` starts from `ExtCount`. The last return value is the value of `ExtCount` plus the number of field extensions that have been introduced, which can be useful for consecutive naming when making a series of resolution calls. A transcendental element will always be displayed as `s`.

Example H116E11

We compute a formal desingularization for the affine surface $z^2 - xy = 0$.

```
> Q := Rationals();
> Qxy<x,y> := PolynomialRing(Q, 2, "glex");
> Qxyz<z> := PolynomialRing(Qxy);
> f := z^2 - x*y;
> desing := ResolveAffineMonicSurface(f); #desing;
3
```

We have computed 3 morphisms. Two of them are centred over the coordinate axes $x = 0$ and $y = 0$. But they might not be of interest, because the surface is normal and has an isolated singularity over the origin.

```
> #ResolveAffineMonicSurface(f : Focus := [x,y]);
1
```

The only remaining morphism corresponds to the exceptional divisor obtained by blowing up the singularity.
Elements in the returned list which define the morphisms of the formal desingularization are examined more closely in the projective surface example below.

```
ResolveProjectiveSurface(S)
```

AdjComp	BoolElt	*Default :* false
ExtName	MonStgElt	*Default :* "*gamma*"
ExtCount	RngIntElt	*Default :* 0
Verbose	Resolve	*Maximum :* 1

The principal function for hypersurface desingularization, similar in description to ResolveAffineMonicSurface above. The argument is either a projective surface $S \subset \mathbf{P}^3_{\mathbf{E}}$ or an irreducible, homogeneous polynomial $s \in \mathbf{E}[x, y, z, w]$ which defines such a surface S. Computes a formal desingularization (see above) of S. It will be a formal desingularization over an automatically chosen subset $C \subset S$ (using again the cylinder over the discriminant curve w.r.t. a nice projection onto some $\mathbf{P}^2_{\mathbf{E}}$). Accordingly the elements of the return list of formal desingularization data are now of the form $((X, Y, Z, W), o)$.

If AdjComp is true, then only a sublist is returned that is still sufficient for the computation of birational invariants and adjoint spaces (see Section 116.3.4). The parameters ExtName and ExtCount and the second return value have the same meaning as in the affine case.

As stated above, the algorithm is based on formally computing a Jung resolution and is described in [Bec07].

Example H116E12

Computing a formal desingularization is easy.

```
> P<x,y,z,w> := PolynomialRing(Rationals(), 4);
> F := w^3*y^2*z+(x*z+w^2)^3;
> desing := ResolveProjectiveSurface(F); #desing;
26
```

Hence, the formal desingularization of the projective surface defined by F contains 26 morphisms. They are represented by tuples of power series that vanish on F. We have a closer look at the first morphism.

```
> prm, ord := Explode(desing[1]);
> IsZero(AlgComb(F, prm)); ord;
true
4
> X, Y, Z, W := Explode(prm);
> Expand(X, 6); Expand(Y, 6); Expand(Z, 6); Expand(W, 6);
true 1
true -s*t^2
true -t^2
true -1/64*s^2*gamma_0*t^5 + 1/2*gamma_0^2*t^3 + gamma_0*t^2 + t
> Domain(W);
Polynomial ring of rank 1 over Algebraic function field defined
over Univariate rational function field over Rational Field
```

```
by $.1^3 - 1/8*s^2
Graded Lexicographical Order
Variables: t
```

One of the morphisms is of type $\mathrm{Spec}\mathbf{Q}(s)[\texttt{gamma\_0}][[t]] \to \mathrm{Proj}\mathbf{Q}[x,y,z,w]/\langle F\rangle$ where $\texttt{gamma\_0}^3 - 1/8s^2 = 0$. In particular, $\mathbf{Q}(s)[\texttt{gamma\_0}]$ is isomorphic to the residue field of the corresponding prime divisor on the desingularization. From this one can for example deduce that it is a rational curve. The morphism is given by the ring homomorphism $x \mapsto 1$, $y \mapsto -st^2$, $z \mapsto -t^2$ and $w \mapsto t + \texttt{gamma\_0}t^2 + 1/2\texttt{gamma\_0}^2t^3 - 1/64s^2\texttt{gamma\_0}t^5 + \ldots$.
The adjoint order for this morphism is 4. Consider the chart $x \neq 0$. The special differential form (see Section 116.3.4) in this chart obtained by dehomogenizing is

$$\frac{x^5}{(\partial F/\partial w)(x,y,z,w)}\mathrm{d}y/x \wedge \mathrm{d}z/x.$$

Substituting the values `X`, `Y`, `Z` and `W` we see that it is mapped to

$$\frac{\mathtt{X}^5}{(\partial F/\partial w)(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})}\mathrm{d}\mathtt{Y}/\mathtt{X} \wedge \mathrm{d}\mathtt{Z}/\mathtt{X}$$

$$= \frac{1}{(\partial F/\partial w)(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})}\mathrm{d}(-st^2) \wedge \mathrm{d}(-t^2)$$

$$= \frac{1}{(\partial F/\partial w)(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})}(2st\mathrm{d}t + t^2\mathrm{d}s) \wedge 2t\mathrm{d}t$$

$$= \frac{1}{(\partial F/\partial w)(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})}2t^3\mathrm{d}s \wedge \mathrm{d}t$$

The adjoint order is minus the overall order of this expression, hence, -3 plus the order of $(\partial F/\partial w)(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})$. We check the computation.

```
> Order(AlgComb(Derivative(F,w), prm));
7
```

If we needed the formal desingularization only in order to compute birational invariants or adjoint spaces we could set the parameter `AdjComp` and forget about some morphisms.

```
> #ResolveProjectiveSurface(F : AdjComp := true);
18
```

116.3.4 Adjoint Systems and Birational Invariants

In this section we describe computation of adjoint spaces. Let $S \subset \mathbf{P}^3_{\mathbf{E}}$ be a surface defined by a homogeneous irreducible polynomial $F \in \mathbf{E}[x_0, x_1, x_2, x_3]$ of degree d and $\Omega_{\mathbf{E}(S)|\mathbf{E}}$ the vector space of rational differential forms of the function field (over the ground field $\mathbf{E}$ of characteristic zero). We can consider $\Omega_{\mathbf{E}(S)|\mathbf{E}}$ a constant sheaf of $\mathcal{O}_S$-modules. Let $U_i \subset S$ be the affine open subsets of the standard covering w.r.t. this choice of variables.

Let $\omega^0_S \subset \Omega^{\wedge 2}_{\mathbf{E}(S)|\mathbf{E}}$ be the subsheaf which is locally generated on U_i by

$$\left(\frac{\partial F/\partial x_j}{x_i^{d-1}}\right)^{-1} \bigwedge_{k\in\{0,\dots,3\}\setminus\{i,j\}} \mathrm{d}\frac{x_k}{x_i}$$

(for an arbitrary choice of $j \neq i$). By sending this generator to x_i^{d-4} one finds that $\omega^0_S \cong \mathcal{O}_S(d-4)$. Further let $F_{S,m} \subset (\Omega^{\wedge 2}_{\mathbf{E}(S)|\mathbf{E}})^{\otimes m}$ be the subsheaf of those forms whose pullbacks are regular on some desingularization of S. It is called the *sheaf of m-adjoints*. It is in fact well-defined, *i.e.*, doesn't depend on any specific desingularization, and one can show $F_{S,m} \subseteq (\omega^0_S)^{\otimes m}$. For more details we refer to [BS08].

Now since $F_{S,m}$ is a coherent sheaf on the projective scheme $S \subset \mathbf{P}^3_{\mathbf{E}}$ it can be defined by its associated graded module $M_{S,m}$ and by the above discussion $F_{S,m}$ is isomorphic to a subsheaf of $\mathcal{O}_S(m(d-4))$. The module $M_{S,m}$ is thus naturally a graded submodule of $(\mathbf{E}[x_0, x_1, x_2, x_3]/\langle F\rangle)(m(d-4))$. The n-th graded piece of $M_{S,m}$, a linear subsytem of the standard linear system of degree $n+m(d-4)$ homogeneous polynomials on S, corresponds to global sections of the Serre twist $F_{S,m}(n)$. This, under pullback, corresponds to the space of global sections of the twisted m-adjoint sheaf $(\omega_X)^{\otimes m}(n)$ for any desingularization X of S, where (n) now signifies twisting by the n-th tensor power of $\mathcal{L}$, the invertible sheaf on X which gives the map into projective space projecting X down onto S.

These adjoint linear systems immediately give the plurigenera of any desingularization X as well as an explicit representation of the important twisted m-adjoint maps into projective space as rational maps from S (defined by the sequence of homogeneous polynomials forming a basis of the adjoint system). These maps are used to take any rational hypersurface to a standard model, as described in the next section.

All functions in this section (and several in the following sections) allow the user to enter precomputed formal desingularization data. It is a good idea to do this if performing several operations on the same hypersurface to avoid repeated computation of this desingularization.

```
HomAdjoints(m,n,S)
```

FormalDesing	SeqEnum	*Default :* 0
Verbose	Classify	*Maximum :* 1

Given a surface S of degree d in $\mathbf{P}^3$ defined over a number field $\mathbf{E}$ together with integers m and n, the intrinsic returns a basis for the vector space of the degree-n graded summand of the graded ring associated to $F_{S,m}$ (*i.e.*, $\Gamma(S, \mathcal{O}_S(n) \otimes F_{S,m})$) as a subspace of the homogeneous forms in $\mathbf{E}[x_0, x_1, x_2, x_3]$ (the coordinate ring of the $\mathbf{P}^3$ ambient) of degree $n + m(d-4)$ (see above).

The parameter FormalDesing may be set to a precomputed formal desingularization (as returned by ResolveProjectiveSurface). The desingularization passed in can be computed with the AdjComp parameter set to true. The default value for FormalDesing is the integer 0, in which case a formal desingularization needs to be computed during function execution.

The function computes the adjoint space as a linear subspace of homogeneous polynomials of the appropriate degree by using the formal divisor morphisms of the formal desingularization to give additional linear conditions at the singular places of S. This is explained fully in [BS08].

```
GeometricGenusOfDesingularization(S)
```

FormalDesing SeqEnum *Default :* 0

Given a hypersurface S in $\mathbf{P}^3$, the intrinsic returns the geometric genus of (any) desingularization of S. The function just computes the dimension of the $(1, 0)$ adjoint space.

As in the case of HomAdjoints, a precomputed desingularization (of S or its defining polynomial) can be passed in via the FormalDesing parameter.

```
PlurigenusOfDesingularization(S,m)
```

FormalDesing SeqEnum *Default :* 0

Given a hypersurface S in $\mathbf{P}^3$, the intrinsic returns the m-th plurigenus of (any) desingularization, X, of S. This is the dimension of the global sections of the sheaf $(\omega_X)^{\otimes m}$ and is just computed as the dimension of the $(m, 0)$ adjoint space.

As for HomAdjoints, a precomputed desingularization (of S or its defining polynomial) can be passed in via the FormalDesing parameter.

```
ArithmeticGenusOfDesingularization(S)
```

FormalDesing SeqEnum *Default :* 0

Given a hypersurface S in $\mathbf{P}^3$, the intrinsic returns the arithmetic genus of (any) desingularization of S. This is computed from a simple formula involving the dimensions of the (1,1)- and (1,2)-adjoints coming from the Riemann-Roch theorem.

As for HomAdjoints, a precomputed desingularization (of S or its defining polynomial) can be passed in via the FormalDesing parameter.

Example H116E13

We compute several adjoint spaces of a surface. We precompute a formal desingularization and pass it to the calls to HomAdjoints.

```
> P<x,y,z,w> := ProjectiveSpace(Rationals(), 3);
> F := w^3*y^2*z+(x*z+w^2)^3;
> S := Surface(P,F);
> desing := ResolveProjectiveSurface(S : AdjComp := true);
> HomAdjoints(1, 0, S : FormalDesing := desing);
[]
```

```
> HomAdjoints(1, 1, S : FormalDesing := desing);
[
    x*z*w + w^3
]
> HomAdjoints(1, 2, S : FormalDesing := desing);
[
    x^2*z^2 - w^4, x^2*z*w + x*w^3, x*y*z*w, x*z^2*w + z*w^3,
    x*z*w^2 + w^4, y*z*w^2, y*w^3
]
> HomAdjoints(1, 3, S : FormalDesing := desing);
[
    x^3*z^2 - x*w^4, x^2*y*z^2, x^2*z^3 - z*w^4,
    x^3*z*w + x^2*w^3, x^2*y*z*w, x*y^2*z*w, x^2*z^2*w - w^5,
    x*y*z^2*w, x*z^3*w + z^2*w^3, x^2*z*w^2 + x*w^4, x*y*z*w^2,
    y^2*z*w^2, x*z^2*w^2 + z*w^4, y*z^2*w^2, x*y*w^3, y^2*w^3,
    x*z*w^3 + w^5, y*z*w^3, y*w^4
]
>
> HomAdjoints(2, 0, S : FormalDesing := desing);
[]
> HomAdjoints(2, 1, S : FormalDesing := desing);
[]
> HomAdjoints(2, 2, S : FormalDesing := desing);
[
    x^2*z^2*w^2 + 2*x*z*w^4 + w^6
]
> HomAdjoints(2, 3, S : FormalDesing := desing);
[
    x^3*z^2*w^2 + 2*x^2*z*w^4 + x*w^6, x^2*y*z^2*w^2 - y*w^6,
    x^2*z^3*w^2 + 2*x*z^2*w^4 + z*w^6,
    x^2*z^2*w^3 + 2*x*z*w^5 + w^7, x*y*z^2*w^3 + y*z*w^5,
    x*y*z*w^4 + y*w^6, y^2*z*w^4
]
```

116.3.5 Classification and Parameterization of Rational Surfaces

This section contains functions for the recognition of rational surfaces in $\mathbf{P}^3$, the classification and transformation to standard models using appropriate m-adjunction maps and, finally, special case code for these standard models, to determine a parametrization of the original hypersurface.

A *non-singular* surface in $\mathbf{P}^r$ with $r \geq 4$ may be transformed to a standard model using the intrinsic MinimalModelRationalSurface (see Section 116.2.6).

```
IsRational(X)
```

`FormalDesing`	SeqEnum	*Default :* 0
`CheckADE`	BoolElt	*Default :* `false`

Returns `true` if the ordinary projective surface X is (geometrically) rational, i.e. birationally isomorphic to the projective plane over the algebraic closure of its base field. This simply uses the Castelnuevo criterion that X is rational if and only if both the arithmetic genus and second plurigenus of any desingularization are zero.

If the ambient of X is $\mathbf{P}^3$ over a number field, there is no assumption about the singularity of X and a formal desingularization will be used to compute plurigenera. Otherwise, X should have at worst simple (A-D-E) singularities and the algorithms of early sections are used for the computations. In this latter case, the singularity status is assumed by default. To force a check for only A-D-E singularities, the user should set the parameter `CheckADE` to `true`. This can be a very heavy verification in higher dimensional ambients.

The computation in the former case uses a formal desingularization of X. To avoid recalculation, a precomputed formal desingularization can be supplied using `FormalDesing` parameter, as with the `HomAdjoints` intrinsic.

116.3.6 Reduction to Special Models

In this section, we describe the function for the birational transformation over the base field of a rational hypersurface in $\mathbf{P}^3$ into a special model of one of the types in the standard classification as listed by Josef Schicho in [Sch98]. He enumerates 5 basic cases [Sch98, p. 17 and Lem. 5.2-5.7] and splits the last case in two, choosing labels "1", "2", "3", "4", "5A" and "5B". The lemmas describing cases 3 and 5A involve a further case distinction. Also, a label "0" is useful for the non-rational case. From this, we get the label set

$$\mathcal{L} := \{\text{“0”}, \text{“1”}, \text{“2”}, \text{“3a”}, \text{“3b”}, \text{“4”}, \text{“5Aa”}, \text{“5Ab”}, \text{“5Ac”}, \text{“5B”}\}$$

Let S be a surface in $\mathbf{P}^3$. In each of the above cases the author specifies a set of adjoint spaces defining interesting maps, either birationally to a special surface or to a rational normal curve (giving a pencil of rational curves on the surface). More precisely, the maps can be computed using $V_{n,m} :=$ `HomAdjoints` (m, n, S) for certain choices of m and n. Let μ to be the smallest integer s.t. $V_{1,\mu+1} \neq [\ ]$. Then the important $V_{n,m}$ for the different cases are as follows:

0	1	2	3a	3b	4
[ ]	$[V_{1,\mu}]$	$[V_{1,\mu}]$	$[V_{2,2\mu+1}, V_{1,\mu}]$	$[V_{2,2\mu+1}]$	$[V_{1,\mu}, V_{2,2\mu-1}]$

5Aa	5Ab	5Ac	5B
$[V_{1,\mu-1}]$	$[V_{1,\mu-1}, V_{2,2\mu-2}]$	$[V_{1,\mu-1}, V_{2,2\mu-2}, V_{3,3\mu-3}]$	$[V_{2,1}]$

The function to find a parameterization of a rational hypersurface, which uses the reduction function below as a first stage, is described in the next section.

`ClassifyRationalSurface(S)`

`FormalDesing` SeqEnum *Default :* 0

Verbose `Classify` *Maximum :* 1

Given an ordinary projective surface S in $\mathbf{P}^3$ over a number field, the intrinsic returns the special rational surface type to which S is birationally equivalent and associated data. If S is not (geometrically) rational, the return values are S itself, a list containing only the identity map on S and the string "Not rational".

If S is rational and Schicho's algorithm reduces it to a standard surface Y over the base field k, Y is the first return value. The second return value is a list of one or two scheme maps. The first is always a birational map from S to Y. There is a second map if and only if Y is a rational scroll or conic bundle. Then S (and Y) have fibration maps to a rational normal curve such that the general fibre is a rational curve (and a line or conic for Y). The fibration map on S is the second return value. Note that, if the base field is $\mathbf{Q}$, in these cases, S can be parameterized by calling `ParametrizePencil` with the fibration map as argument.

The third return value is a string describing the type of Y. It is "P2" (for the projective plane!), "Quadric surface" (for a degree 2 surface in $\mathbf{P}^3$), "Rational scroll", "Conic bundle" or "Del Pezzo of degree d" where $1 \leq d \leq 9$. The Del Pezzos might be degenerate (with simple singularities) and are anticanonically embedded in $\mathbf{P}^d$ except for degrees 1 and 2 when they have their standard weighted-projective embeddings.

As usual, to avoid recalculation, a precomputed formal desingularization can be given using the `FormalDesing` parameter, as in the case of the `HomAdjoints` intrinsic.

Example H116E14

Here are a few examples.

```
> P<x,y,z,w> := ProjectiveSpace(Rationals(),3);
```

The first surface:

```
> p1 := x^4 + y^4 - z^2*w^2;
> _,_,typ := ClassifyRationalSurface(Surface(P,p1));
> typ;
Not rational
```

The second surface:

```
> p2 := 2*x + y + 8*z + 5*w;
> Y,_,typ := ClassifyRationalSurface(Surface(P,p2));
> typ; Y;
P^2
Surface over Rational Field defined by
0
```

The third surface:

```
> p3 := x^2 - 4*x*z + 3*x*w + y*z - y*w + 2*z^2 - 3*z*w + w^2;
> _,_,typ := ClassifyRationalSurface(Surface(P,p3));
> typ;
Quadric surface
```

The fourth surface:

```
> p4 := (y^2 - w*z)*(w^2 - y*x) + (x*z - y*w)^2;
> S := Surface(P,p4);
> Y,mps,typ := ClassifyRationalSurface(S);
> typ;
Rational scroll
> mps[2]; // the fibration map
Mapping from: Srfc: S to Scheme over Rational Field defined by
$.1*$.2 - $.3^2
with equations :
x*y - w^2
y^2 - z*w
x*z - y*w
```

The fifth surface:

```
> p5 := x^3*y - 4*x^3*z - 6*x^3*w - 3*x^2*y^2 - 2*x^2*y*z
>     - 3*x^2*y*w + 50*x^2*z^2 + 146*x^2*z*w + 108*x^2*w^2
>     - 11*x*y^2*z + 2*x*y^2*w + 61*x*y*z^2 + 149*x*y*z*w
>     + 65*x*y*w^2 + 68*x*z^3 + 228*x*z^2*w + 260*x*z*w^2
>     + 112*x*w^3 + 4*y^4 - 13*y^3*z - 19*y^3*w + 20*y^2*z^2
>     + 77*y^2*z*w + 55*y^2*w^2 + 40*y*z^3 + 106*y*z^2*w
>     + 58*y*z*w^2 - 2*y*w^3 + 22*z^4 + 84*z^3*w + 130*z^2*w^2
>     + 108*z*w^3 + 38*w^4;
> S := Surface(P,p5);
> _,mps,typ := ClassifyRationalSurface(S);
> typ;
P2
> mps[1]; //birational map from Y to P2
Mapping from: Srfc: S to Surface over Rational Field defined by
0
with equations :
x^2-1114/45*x*z-232/15*y*z-241/15*z^2-1543/45*x*w-319/15*y*w-1327/45*z*w-
    457/45*w^2
x*y-182/45*x*z-11/15*y*z-38/15*z^2-284/45*x*w-17/15*y*w-266/45*z*w-146/45*w^2
y^2-16/45*x*z-28/15*y*z-4/15*z^2-22/45*x*w-61/15*y*w+32/45*z*w+92/45*w^2
```

The sixth surface:

```
> p6 := x^2*y^2 + 8*x^3*y + 4*x^4 + x*y*z^2 - x^2*z^2 - y^2*w^2
>     - 7*x*y*w^2 + 8*x^2*w^2;
> S := Surface(P,p6);
> Y,mps,typ := ClassifyRationalSurface(S);
> typ; Y;
Conic bundle
Surface over Rational Field defined by
```

```
$.1^2 + 2*$.1*$.2 + 1/4*$.1*$.3 - 1/4*$.4^2 + 1/4*$.4*$.5 +
    2*$.6^2 - 7/4*$.6*$.7 - 1/4*$.7^2,
$.2^2 - $.1*$.3,
$.2*$.4 - $.1*$.5,
$.3*$.4 - $.2*$.5,
$.2*$.6 - $.1*$.7,
$.3*$.6 - $.2*$.7,
$.5*$.6 - $.4*$.7
> mps[2]; //the fibration map
Mapping from: Srfc: S to Projective Space of dimension 1 over Rational Field
Variables: $.1, $.2
with equations :
x
y
```

The seventh surface:

```
> p7 := x^2*w^3 + y^3*w^2 + z^5;
> Y,_,typ := ClassifyRationalSurface(Surface(P,p7));
> typ; Y; Ambient(Y);
Del Pezzo degree 1
Surface over Rational Field defined by
$.1^5*$.2 + $.3^3 + $.4^2
Projective Space of dimension 3 over Rational Field
Variables: $.1, $.2, $.3, $.4
The grading is:
    1, 1, 2, 3
```

The seventh surface:

```
> p8 := w^3*y^2*z + (x*z + w^2)^3;
> Y,_,typ := ClassifyRationalSurface(Surface(P,p8));
> typ; Y;
Del Pezzo degree 6
Surface over Rational Field defined by
$.1^2 - 4*$.5^2 + $.3*$.6 - 3*$.6*$.7,
$.1*$.2 + 2*$.2*$.5 + $.3*$.7,
$.1*$.4 + 2*$.4*$.5 + $.6^2,
$.2*$.4 + $.5^2 + $.6*$.7,
$.3*$.4 - $.1*$.6 - $.4*$.7,
$.1*$.5 + 2*$.5^2 + $.6*$.7,
$.3*$.5 - $.1*$.7 - $.5*$.7,
$.2*$.6 - $.1*$.7 - $.5*$.7,
$.5*$.6 - $.4*$.7
```

116.3.7 Parametrization of Rational Surfaces

The package also includes functions to directly parametrize rational surfaces (over $\mathbf{Q}$). These use the reduction to special type, described in the last section, followed by specialised algorithms for the special cases, which are described in the following sections and the section on Del Pezzo surfaces. There is a version for hypersurfaces in $\mathbf{P}^3$ and one for more general rational surfaces which are first birationally projected to a hypersurface. Note, however, that the projection method can be very inefficient because it often introduces nasty singularities that cause the resolution process to hang. If it is known that the surface S is non-singular, it is usually much better to use `MinimalModelRationalSurface` to get a birational map from S to a standard model Y and call the relevant special case parametrization routine for Y directly.

`ParametrizeProjectiveHypersurface(X, P2)`

`FormalDesing`	SeqEnum	*Default :* 0
Verbose	`Classify`	*Maximum :* 1

Given a surface X in $\mathbf{P}^3_{\mathbf{Q}}$ and a projective plane $P2$ over $\mathbf{Q}$, the intrinsic returns `false` if the surface is not rational over $\mathbf{Q}$, otherwise returns `true` and a birational parameterization $P2 \to X$.

The function begins by mapping X, as in `ClassifyRationalSurface`, to a surface of special type. As for that function, if a formal desingularization (for the defining polynomial p of X) is already known, it can be passed as parameter `FormalDesing`.

It is assumed that X is defined over $\mathbf{Q}$ because some of the special type routines assume this, partly for simplicity. This will probably be generalised in future releases.

`ParametrizeProjectiveSurface(X, P2)`

Verbose	`Classify`	*Maximum :* 1

Given an ordinary projective surface X in $\mathbf{P}^n_{\mathbf{Q}}$ for some $n \geq 2$ and a projective plane $P2$ over $\mathbf{Q}$, returns `false` if the surface is not rational over $\mathbf{Q}$, otherwise return `true` and a birational parametrization $P2 \to X$.

The function finds a birational projection to a hypersurface in $\mathbf{P}^3$ and then calls `ParametrizeProjectiveHypersurface`.

It should be noted that the birational projection may produce a very singular hypersurface defined by a polynomial with large coefficients. Then, the desingularisation, classification and special parameterization routines may each be very slow. As noted above, it may be better to try to use `MinimalModelRationalSurface` followed by one of the specialised parametrization routines for a non-singular rational X for larger n.

Example H116E15

We try to parameterize the hypersurfaces given by polynomials `p1 - p8` from the previous example.

The surface defined by $p1$:

```
> P2<X,Y,W> := ProjectiveSpace(Rationals(), 2);
> ParametrizeProjectiveHypersurface(Surface(P, p1), P2);
false
```

The surface defined by $p2$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p2), P2);
true Mapping from: Prj: P2 to Surface over Rational Field
defined by 2*x + y + 8*z + 5*w
with equations :
-1/2*X - 4*Y - 5/2*W
X
Y
W
```

The surface defined by $p3$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p3), P2);
true Mapping from: Prj: P2 to Surface over Rational Field
defined by x^2 - 4*x*z + 3*x*w + y*z - y*w + 2*z^2 - 3*z*w + w^2
with equations :
X^2 - 2*X*W
X^2 + X*Y - 4*X*W - Y*W + 2*W^2
X^2 - 3*X*W + Y*W
X^2 - 4*X*W + Y*W + 2*W^2
```

The surface defined by $p4$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p4), P2);
true Mapping from: Prj: P2 to Surface over Rational Field
defined by x^2*z^2 - x*y^3 - x*y*z*w + 2*y^2*w^2 - z*w^3
with equations :
2*X^2*Y^2*W^2 - Y*W^5
X^4*Y^2 - X^2*Y*W^3
X^3*Y*W^2
X^3*Y^2*W
```

The surface defined by $p5$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p5), P2);
true Mapping from: Prj: P2 to Surface over Rational Field
defined by ...
with equations :
-1/4*X^2 + 9/2*X*Y + 1/2*X*W - 69/4*Y^2 - 87/8*Y*W + 1/2*W^2
1/2*X*Y + 11/4*X*W - 5/8*Y^2 - 131/8*Y*W - 63/8*W^2
-11/8*X*Y + 7/8*X*W + 13/4*Y^2 - 11/2*W^2
X*Y - 1/8*X*W - 23/8*Y^2 + 4*W^2
```

The surface defined by $p6$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p6), P2);
```

```
true Mapping from: Prj: P2 to Surface over Rational Field
defined by 4*x^4 + 8*x^3*y + x^2*y^2 - x^2*z^2 + 8*x^2*w^2
            + x*y*z^2 - 7*x*y*w^2 - y^2*w^2
with equations :
-3/8*X^2*Y^2*W - 2*X^2*Y*W^2 - 8/3*X^2*W^3 - 59/24*X*Y^2*W^2
    - 38/3*X*Y*W^3 - 16*X*W^4 + 17/6*Y^2*W^3 + 44/3*Y*W^4
    + 56/3*W^5
-3/8*X^3*Y^2 - 2*X^3*Y*W - 8/3*X^3*W^2 - 59/24*X^2*Y^2*W
    - 38/3*X^2*Y*W^2 - 16*X^2*W^3 + 17/6*X*Y^2*W^2
    + 44/3*X*Y*W^3 + 56/3*X*W^4
-1/8*X^2*Y^2*W - 4/3*X^2*Y*W^2 - 8/3*X^2*W^3 - 1/12*X*Y^2*W^2
    - 16/3*X*Y*W^3 - 40/3*X*W^4 + 19/3*Y^2*W^3 + 104/3*Y*W^4
    + 48*W^5
-3/8*X^2*Y^2*W - 2*X^2*Y*W^2 - 8/3*X^2*W^3 - 8/3*X*Y^2*W^2
    - 14*X*Y*W^3 - 56/3*X*W^4 + Y^2*W^3 + 20/3*Y*W^4 + 32/3*W^5
```

The surface defined by $p7$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p7), P2);
true Mapping from: Prj: P2 to Surface over Rational Field defined by
z^5 + y^3*w^2 + x^2*w^3
with equations :
-X^362*Y^144*W^79 - 9*X^363*Y^141*W^81 - 36*X^364*Y^138*W^83 -
    84*X^365*Y^135*W^85 - 126*X^366*Y^132*W^87 - 126*X^367*Y^129*W^89 -
    84*X^368*Y^126*W^91 - 36*X^369*Y^123*W^93 - 9*X^370*Y^120*W^95 -
    X^371*Y^117*W^97
X^359*Y^148*W^78 + 10*X^360*Y^145*W^80 + 45*X^361*Y^142*W^82 +
    120*X^362*Y^139*W^84 + 210*X^363*Y^136*W^86 + 252*X^364*Y^133*W^88 +
    210*X^365*Y^130*W^90 + 120*X^366*Y^127*W^92 + 45*X^367*Y^124*W^94 +
    10*X^368*Y^121*W^96 + X^369*Y^118*W^98
-X^357*Y^150*W^78 - 11*X^358*Y^147*W^80 - 55*X^359*Y^144*W^82 -
    165*X^360*Y^141*W^84 - 330*X^361*Y^138*W^86 - 462*X^362*Y^135*W^88 -
    462*X^363*Y^132*W^90 - 330*X^364*Y^129*W^92 - 165*X^365*Y^126*W^94 -
    55*X^366*Y^123*W^96 - 11*X^367*Y^120*W^98 - X^368*Y^117*W^100
X^354*Y^153*W^78 + 12*X^355*Y^150*W^80 + 66*X^356*Y^147*W^82 +
    220*X^357*Y^144*W^84 + 495*X^358*Y^141*W^86 + 792*X^359*Y^138*W^88 +
    924*X^360*Y^135*W^90 + 792*X^361*Y^132*W^92 + 495*X^362*Y^129*W^94 +
    220*X^363*Y^126*W^96 + 66*X^364*Y^123*W^98 + 12*X^365*Y^120*W^100 +
    X^366*Y^117*W^102
and inverse
z^4*w^8
y*z^2*w^9
x*z*w^10
```

The surface defined by $p8$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p8), P2);
true Mapping from: Prj: P2 to Surface over Rational Field defined by
x^3*z^3 + 3*x^2*z^2*w^2 + 3*x*z*w^4 + y^2*z*w^3 + w^6
with equations :
```

```
2*X^20*Y^3*W - 4*X^19*Y^3*W^2 + 4*X^17*Y^3*W^4 - 2*X^16*Y^3*W^5
4*X^19*Y^4*W - 8*X^18*Y^4*W^2 + 4*X^17*Y^4*W^3
1/2*X^16*Y^7*W - X^15*Y^7*W^2 + 1/2*X^14*Y^7*W^3
-X^18*Y^5*W + 3*X^17*Y^5*W^2 - 3*X^16*Y^5*W^3 + X^15*Y^5*W^4
> IsRational(Surface(P, p7));
true
```

Here are two easy examples of parameterizing non-hypersurfaces.

```
> P4<u,v,w,x,y> := ProjectiveSpace(Rationals(),4);
> P2<X,Y,Z> := ProjectiveSpace(Rationals(), 2);
> S := Surface(P4,[u^2 + v^2 + w^2 - x^2, y - x]);
> ParametrizeProjectiveSurface(S, P2);
true Mapping from: Prj: P2 to Srfc: S
with equations :
-2*X*Z + 2*Z^2
-2*Y*Z
X^2 + Y^2 - 2*X*Z
-X^2 - Y^2 + 2*X*Z - 2*Z^2
-X^2 - Y^2 + 2*X*Z - 2*Z^2
and inverse
u + w + y
v
w + x
```

Here we parametrize a particularly easy surface – $\mathbf{P}^2$ itself!

```
> S := Surface(P2,[]);
> ParametrizeProjectiveSurface(S, P2);
true Mapping from: Prj: P2 to Srfc: S
with equations :
X
Y
Z
and inverse
X
Y
Z
```

```
Solve(p, F)
```

For convenience, this intrinsic provides a purely algebraic version of parameterization of an affine hypersurface.

Given a polynomial $p \in \mathbf{Q}[x, y, z]$, the equation of a (not necessarily irreducible affine hypersurface S) and a two-variable rational function field $F = \mathbf{Q}(u, v)$, the function finds birational parameterizations of the irreducible components of S (that are parametrizable over $\mathbf{Q}$).

A sequence of triples $(X, Y, Z) \in F^3$ is returned such that $p(X, Y, Z) = 0$ and each triple gives an isomorphism of F to the function field of a component of S.

The routine may again result in a runtime error if it involves parameterizations of special surface types that are not yet implemented, as for the preceding functions.

Example H116E16

The following affine hypersurface has three irreducible factors: one not rational and two that are rational and parameterizable over **Q**.

```
> Q := RationalField();
> P<x,y,z> := PolynomialRing(Q, 3);
> F<s,t> := RationalFunctionField(Q, 2);
> p := (x^4+y^4-z^2)*(2*x + y + 8*z + 5)
>       *(x^2 - 4*x*z + 3*x + y*z - y + 2*z^2 - 3*z + 1);
> Solve(p, F);
[
    [ -1/2*s - 4*t - 5/2, s, t ],
    [
        (s^2 - 2*s)/(s^2 - 4*s + t + 2),
        (s^2 + s*t - 4*s - t + 2)/(s^2 - 4*s + t + 2),
        (s^2 - 3*s + t)/(s^2 - 4*s + t + 2)
    ]
]
```

116.3.8 Parametrization of Special Surfaces

In this section we describe routines for the explicit parameterization of the special classes of rational surfaces that arise from the reduction of the general case via m-adjoint maps. The algorithms are the work of Josef Schicho, in collaboration with others in some cases. The functions are used in the general parameterization routines but can also be called directly by the user.

`ParametrizeQuadric(X,P2)`

Suppose the scheme $X \subset \mathbf{P}^3_{\mathbf{Q}}$ is a geometrically irreducible quadric (degree 2) projective hypersurface and $P2$ is a projective plane. The intrinsic returns `false` if X is not parametrizable over the rationals, otherwise it returns `true` together with a birational parameterization $P2 \to X$.

Given a rational point p on X, the intrinsic is based on a simple, well-known algorithm (see [Sch98, Sec. 3.1]). Finding the point p is equivalent to finding a non-trivial isotropic vector for F, the quadric form in four variables defining X. This is achieved by a reduction to the solution of two quadrics in three variables, which is performed using standard lattice methods. The solubility routines here assume that the quadric is defined over **Q**.

Example H116E17

We give a few simple examples.

```
> Q := Rationals();
> P2<X,Y,W> := ProjectiveSpace(Q, 2);
> P3<x,y,z,w> := ProjectiveSpace(Q, 3);
> X1 := Scheme(P3, x^2 + y^2 + z^2 + w^2);
> X2 := Scheme(P3, x^2 + y^2 + z^2 - w^2);
> X3 := Scheme(P3, x^2 + y^2 + z^2);
> X4 := Scheme(P3, x^2 + y^2 - z^2);
> X5 := Scheme(P3, x^2 - 4*x*z + 3*x*w + y*z - y*w + 2*z^2
>                   - 3*z*w + w^2);
> ParametrizeQuadric(X1, P2);
false
> ParametrizeQuadric(X2, P2);
true Mapping from: Prj: P2 to Sch: X2
with equations :
2*X*W
2*Y*W
-X^2 - Y^2 + W^2
X^2 + Y^2 + W^2
> ParametrizeQuadric(X3, P2);
false
> ParametrizeQuadric(X4, P2);
true Mapping from: Prj: P2 to Sch: X4
with equations :
X*Y
-1/2*X^2 + 1/2*Y^2
1/2*X^2 + 1/2*Y^2
Y*W
> ParametrizeQuadric(X5, P2);
true Mapping from: Prj: P2 to Sch: X5
with equations :
X^2 - 2*X*W
X^2 + X*Y - 4*X*W - Y*W + 2*W^2
X^2 - 3*X*W + Y*W
X^2 - 4*X*W + Y*W + 2*W^2
```

```
ParametrizePencil(phi, P2)
```

Let X be an ordinary projective birationally ruled surface, given as the domain of a rational pencil ϕ defined over $\mathbf{Q}$ (*i.e.*, a rational map $X \to \mathbf{P}^n_{\mathbf{Q}}$ for some n with image a rational normal curve) and $P2$ is a projective plane over $\mathbf{Q}$. The intrinsic returns `false` if X is not parameterizable over the rationals. Otherwise, it returns `true` and a birational parameterization $P2 \to X$.

These intrinsics take care of rational scrolls and conic bundles in the classification of special surfaces. The algorithm is described in [Sch00]. The scheme X is not assumed to be non-singular.

Example H116E18

We start with a (singular) degree 4 hypersurface X in P^3 and construct a pencil from P^2 to X.

```
> Q := Rationals();
> P3<x,y,z,w> := ProjectiveSpace(Q, 3);
> P2<X,Y,Z> := ProjectiveSpace(Q, 2);
> X := Scheme(P3, x^2*z^2 - x*y^3 - x*y*z*w + 2*y^2*w^2 - z*w^3);
> pencil := map<X -> P2 | [x*y - w^2, y^2 - z*w, x*z - y*w]>;
> DefiningPolynomial(Image(pencil));
X*Y - Z^2
> ParametrizePencil(pencil, P2);
true Mapping from: Prj: P2 to Sch: X
with equations :
-X^4*Y + 2*X^2*Z^3
-X^2*Y*Z^2 + Z^5
X*Y*Z^3
X*Z^4
```

```
ParametrizeDelPezzo(X, P2)
```

The argument X should be a (anticanonically-embedded) Del Pezzo surface (of type `Sch` or `Srfc` and $P2$ a projective plane both defined over $\mathbf{Q}$. The function returns `false` if X is not parametrizable over the rationals and returns `true` and a birational parameterization $P2 \to X$ otherwise.

This intrinsic is the main interface to a suite of functions parameterizing Del Pezzo surfaces (over $\mathbf{Q}$). These include the anticanonically Del Pezzos of degrees 1 and 2, which lie in non-trivially weighted projective spaces. A degree d Del Pezzo surface refers to a rational surface that is embedded in projective space by its anticanonical divisor. For degrees 1 and 2 this means an *ample* embedding into weighted projective space. For $3 <= d <= 9$, this is a very-ample embedding giving X as a degree d surface in ordinary projective space of dimension d.

It should be noted that not only do the routines handle the usual non-singular cases, but they also deal with degenerate singular cases (arising in degrees $d >= 3$. In the latter case, rather than blowing up $9 - d$ *distinct* points in the plane, some of the blown-up points are “infinitely near” points: lying on the exceptional curves corresponding to already blown-up points). This is important as these degenerate cases can arise in the general parameterization of hypersurfaces in $\mathbf{P}^3$.

The package contains routines to find and blow down sets (defined over $\mathbf{Q}$) of exceptional curves, reducing to a DelPezzo of degree d, $5 <= d <= 9$. After this reduction, these cases are handled by the Magma routines described in Section 116.4.3 (which now also handle singular Del Pezzos). There is an exception to the above rule. For *singular* degree 3 and degree 4 Del Pezzo surfaces, it is more efficient to apply special case code directly rather than trying to blow down lines to get to higher degree. Starting in V2.17, special functions are provided do this that can be called directly for a singular anti-canonical degree 3 or 4 surface and are described in the next section.

The routines are not yet fully documented. For the location of exceptional curves and the blowing-down, some details may be found in [Sch98, Sec. 3.5] while [Man86] contains the general theory. There are also implementation notes in the appendix of the software documentation report [Bec08] from which this documentation has been adapted. For the nonsingular degrees 6, 8 and 9 cases, which use the Lie algebra method and the degree 5 case, see the references in Section 116.4.3.

116.4 Del Pezzo Surfaces

116.4.1 Introduction

This section contains a collection of geometric and arithmetic routines for Del Pezzo surfaces in their anti-canonical embeddings (the full weighted projective anticanonical embedding for degrees 1 and 2).

There are routines for creation, parametrization, minimisation and reduction, construction, computation of invariants for degree 3, and point-counting for degree 3 surfaces over a finite field.

There is a specialised type for Del Pezzos, `SrfDelPezzo`, which is a subtype of type `Srfc`. Some intrinsics use this type for arguments while some use the more general `Srfc` (or even `Sch`).

116.4.2 Creation of General Del Pezzos

```
DelPezzoSurface(P,L)
```

```
DelPezzoSurface(S)
```

```
DelPezzoSurface(Z)
```

The Del Pezzo surface of degree $9 - d$, embedded by its anticanonical system, which arises by blowing up the projective plane P in the d points. The arguments are either plane P and a list of points L, the set of points Q, or the length d zero-dimensional scheme Z defining the points. If the points are not in sufficiently general position (so that the anticanonical image is not smooth) then an error is reported.

`DelPezzoSurface(f)`

The argument f should be a degree three homogeneous polynomial in a 4-variable polynomial ring P (with grevlex ordering). Creates the degree 3 Del Pezzo surface with defining polynomial f inside $\mathbf{P}^3 = Proj(P)$. If this surface is not smooth an error is reported.

`IsDelPezzo(Y)`

Returns `true` if and only if scheme Y in ordinary projective space is an abstract Del Pezzo surface. If so, it also returns the image X of the standard (pluri-)anticanonical embedding of Y, and the map $Y \to X$. Note that this can be a computationally very heavy function if Y is in a reasonably high-dimensional ambient.

116.4.3 Parametrization of Del Pezzo Surfaces

Del Pezzo surfaces are a special type of non-singular projective surface. A good reference for their general properties is [Man86]. A Del Pezzo surface X has a degree $1 \le d \le 9$. For $d \ge 3$, the standard representation is as a degree d surface in $\mathbf{P}^d$ for which a hyperplane section is an anti-canonical divisor. When we talk about Del Pezzos in this section, we mean a surface in that anti-canonical form.

Del Pezzo surfaces are birationally equivalent to the projective plane $\mathbf{P}^2$ over an algebraically-closed field. That is, there exists an invertible scheme map from $\mathbf{P}^2$ to X : a *parametrization*. Most of the functions in this section are concerned with the existence of parametrizations of X over a number field.

The significance of Del Pezzo surfaces comes from *adjunction theory* (see [SvdV87]). The adjunction map for surfaces is a general construct that contracts certain lines, known as exceptional lines, to points. Repeated application of the adjunction map to a rational surface results in a reduction to a surface in one of a small number of families, including the Del Pezzos. For a non-singular ordinary projective surface Y, the intrinsic `MinimalModelRationalSurface` is now available to produce the birational map from Y to one of the special, terminal cases that include the Del Pezzos.

Hence the parametrization problem for general rational surfaces reduces via adjunction (a purely algebraic construction) to parametrization of surfaces in the specific families, which is an arithmetic problem (ie dependent on the ground field). This is the surface analog of the simpler situation for curves. Any rational curve can be algebraically reduced to the projective line or a plane conic and the parametrization of plane conics is also an arithmetic problem.

Thus, the parametrization of Del Pezzo surfaces is an important component in that of general rational surfaces. General parametrization code for rational hypersurfaces in $\mathbf{P}^3$ is described in the previous section and it makes use of the routines described here.

If X is parametrizable with $d \ge 3$, then we can blow down (contract) exceptional lines on it to arrive at a surface with $d = 5$, 6, 8 or 9. The functions described here deal with parametrization in those cases using computational methods based around the Lie algebra of the automorphism group of X. For the theory behind these algorithms, see

[dG06], [dGP], [HS06] and [GSHPBS12]. In one of the examples, we parametrize a cubic hypersurface (degree 3 Del Pezzo) by blowing down to a degree 6 surface.

Although reduction to degree 9 is always possible, for $d = 7$ it is more efficient to work directly using the Lie algebra method. There is now also an intrinsic for $d = 7$ provided by Josef Schicho. Schicho has also written code for the $d = 5$ case, minimal or not, that uses the more geometrical method described in the above reference and a corresponding intrinsic is provided.

Furthermore, the degree $5 - 8$ intrinsics now also cover degenerate cases of singular Del Pezzos in their anticanonical projective embeddings. The additional code for the singular cases is also due to Josef Schicho.

A feature (from V2.17) is the provision of special case code for degree 3 and degree 4 singular Del Pezzos. For some of these, it is not possible to blow down any exceptional lines over the base field but the surface is still parametrizable. Additionally, it is usually much more efficient to handle these cases directly without blowing down curves to get to higher degree.

A general intrinsic `ParametrizeDelPezzo` for parametrizing any Del Pezzo surface of degree $d \geq 1$ in its anticanonical weighted embedding (for $d = 1, 2$ the anticanonical divisor is no longer very ample, but gives an ample embedding into weighted projective space) is described in the previous section. This blows down exceptional lines to reach degree $d \geq 5$ and then invokes one of the intrinsics described in this section. It is more efficient to call the appropriate intrinsic directly if the starting point is either the $d \geq 5$ case or the $d = 3,\ 4$ singular cases.

```
SetVerbose("ParamDP", v)
```

Set the verbose printing level for the Del Pezzo parametrizing functions. Currently the legal values for v are `true`, `false`, 0, 1 and 2 (`false` is the same as 0, and `true` is the same as 1).

```
ParametrizeDegree9DelPezzo(X)
```

Let X be a degree 9 Del Pezzo surface anticanonically embedded in 9-dimensional projective space. For this function the base field should be $\mathbf{Q}$. The surface X is defined by 27 degree 2 polynomials. The function performs only basic checks that the input X is valid.

If X is parametrizable over $\mathbf{Q}$ then there is a parametrization $\phi : \mathbf{P}^2 \to X$ which is everywhere-defined and given by cubic polynomials in the variables of $\mathbf{P}^2$. The function returns whether such a ϕ exists and, if so, ϕ also.

```
ParametrizeDegree8DelPezzo(X)
```

Let X be a degree 8 Del Pezzo surface anticanonically embedded in 8-dimensional projective space over a number field. The surface X is defined by 20 degree 2 polynomials. The function performs only basic checks that the input X is valid.

For degree 8, there are two types of non-singular Del Pezzo surface, the second type splitting into subfamilies:

1) X is isomorphic to $\mathbf{P}^2$ with a single rational point blown up.

2i) X is isomorphic to $T_1 \times T_2$ with T_i Galois twists of $\mathbf{P}^1$.

2ii) X is isomorphic to a Galois twist of $\mathbf{P}^1 \times \mathbf{P}^1$ where Galois acts transitively on the two $\mathbf{P}^1$ factors.

In case 1), X is always parametrizable.
In case 2i), X is parametrizable $\Leftrightarrow$ both T_i are trivial twists of $\mathbf{P}^1$ $\Leftrightarrow$ X is isomorphic to $\mathbf{P}^1 \times \mathbf{P}^1$.
In case 2ii), there is an infinite family of parametrizable X_a classified by $a \in \mathbf{Q}^*/\mathbf{Q}^{*2}$. The scheme X_a is isomorphic (properly, not just birationally) to the surface in $\mathbf{P}^3$ given by the equation $x_0^2 - ax_1^2 = x_2x_3$.

The main function determines whether X is parametrizable over $\mathbf{Q}$. If so, there is a parametrization $\phi : \mathbf{P}^2 \to X$ (given by cubic polynomials in case 1 and by degree 4 polynomials in case 2) and this is also returned.

The intrinsic also handles the degenerate case of a singular degree 8 Del Pezzo surface. This case is recognised directly from the Lie algebra computation which is part of the main routine and an appropriate adaptation of the general method is used.

Example H116E19

In this example, we parametrize an anticanonical sphere X_2, in the above notation. This is obviously an artificial illustration, as we start with X_2 in the form $F = x_0^2 - 2x_1^2 - x_2x_3 = 0$, which is trivial to parametrize directly! The surface in this form is embedded anticanonically in $\mathbf{P}^8$ by any 9-dimensional vector space complement of $< F >$ in the 10-dimensional linear system of all degree 2 polynomials in $x_0 \dots x_3$. The parametrizing map is undefined at precisely 2 points of the plane. Geometrically, the map consists of a blowup of these 2 points followed by a blowdown of the line joining them.

```
> P3<x0,x1,x2,x3> := ProjectiveSpace(Rationals(),3);
> X2 := Scheme(P3,x0^2-2*x1^2-x2*x3);
> L := LinearSystem(P3,2);
> L := LinearSystemTrace(L,X2);
> P8<x1,x2,x3,x4,x5,x6,x7,x8,x9> := ProjectiveSpace(Rationals(),8);
> X := map<X2->P8|Sections(L)>(X2); X;
Scheme over Rational Field defined by
x1^2 - 2*x4^2 - x4*x8,
x1*x2 - 2*x4*x5 - x5*x8,
x2^2 - 2*x4*x7 - x7*x8,
x1*x3 - 2*x4*x6 - x5*x9,
x2*x3 - 2*x4*x8 - x7*x9,
x3^2 - 2*x4*x9 - x8*x9,
-x1*x5 + x2*x4,
-x1*x6 + x3*x4,
-x1*x7 + x2*x5,
-x1*x8 + x3*x5,
-x4*x7 + x5^2,
-x1*x8 + x2*x6,
-x1*x9 + x3*x6,
-x4*x8 + x5*x6,
-x4*x9 + x6^2,
-x2*x8 + x3*x7,
-x5*x8 + x6*x7,
-x2*x9 + x3*x8,
-x5*x9 + x6*x8,
-x7*x9 + x8^2
> boo,prm := ParametrizeDegree8DelPezzo(X);
> boo;
true
> prm;
Mapping from: Prj: P2 to Sch: X
with equations :
-1/4*U*V*W^2
-1/16*V*W^3
2*U^2*V*W - 4*V^3*W
-1/8*U^2*W^2
-1/32*U*W^3
U^3*W - 2*U*V^2*W
-1/128*W^4
1/4*U^2*W^2 - 1/2*V^2*W^2
-8*U^4 + 32*U^2*V^2 - 32*V^4
> bs := ReducedSubscheme(BaseScheme(prm)); bs;
Scheme over Rational Field defined by
U^2 - 2*V^2,
W
```

```
ParametrizeDegree7DelPezzo(X)
```

Let X be a degree 7 Del Pezzo surface anticanonically embedded in 7-dimensional projective space over a number field. We allow that X can be a degenerate (singular) Del Pezzo surface here. The scheme X is always parametrizable over the base field and this intrinsic returns such a parametrisation without reduction to degree 8 or 9 but directly from the Lie Algebra method.

```
ParametrizeDegree6DelPezzo(X)
```

ExistenceOnly BoolElt *Default :* false

Let X be a degree 6 Del Pezzo surface anticanonically embedded in 6-dimensional projective space. For this function the base field K may be $\mathbf{Q}$ or a number field. The surface X is defined by nine degree 2 polynomials. The function performs only basic checks that the input X is valid. **NB:** This intrinsic only handles the non-singular case. For a singular (degenerate) degree 6 Del Pezzo, use the intrinsic that follows.

The connected component of the automorphism group of X is a 2-dimensional torus over K. For any of the possible tori, there is a family of degree 6 Del Pezzos which correspond to principal homogeneous spaces of the torus up to isomorphism.

The parametrizability of X is equivalent to X corresponding to the trivial homogeneous space of its torus.

The function determines whether a parametrization $\phi : \mathbf{P}^2 \to X$ exists over K and returns one when this is the case. The degree of the polynomials defining a "minimal" ϕ (one which is undefined at the smallest number of points) is 3,4 or 6 depending on the torus type. The parametrization returned is always of this minimal degree.

The surface X is parametrizable if and only if it contains a point over K. Furthermore, it satisfies the local-global principle: it has a point over $K \Leftrightarrow$ it has a point over each p-adic completion of K. (These statements are also true for degree 8 and 9 Del Pezzos)

The `ExistenceOnly` option allows the function to just perform this local solubility check, deciding upon the existence of a parametrization without explicitly constructing one. Depending on the torus type, simultaneous norm equations over a degree 6 field extension of K or a single norm equation over a degree 3 extension of K may have to be solved to construct a parametrization. This is a hard computation, especially if K is not $\mathbf{Q}$, whereas the pure existence check is quite fast.

```
Degree6DelPezzoType2_1(K,pt)
```

```
Degree6DelPezzoType2_2(K,pt)
```

```
Degree6DelPezzoType2_3(K,pt)
```

```
Degree6DelPezzoType3(K,pt)
```

```
Degree6DelPezzoType4(K,K1,pt)
```

```
Degree6DelPezzoType6(K,pt)
```

These functions generate the parametrizable degree 6 Del Pezzo surface X whose (connected) automorphism group is the torus T, which comes from field data K, and which contains point *pt*.

The point *pt* must be in 6-dimensional projective space over the base field k of a number field K. Its first projective coordinate may not be 0 and, depending on the torus type, certain of its other coordinates must also be non-zero.

The torus types and corresponding fields K for the various functions are as follows ($pt = [a_0, \ldots, a_6]$):

`Type2_1`. K/k should be a quadratic extension. $T(k) = K^*$ and T acts on $\mathbf{P}^6$ to give an X with degree 3 minimal parametrization. *pt* satisfies not($a_1 = a_2 = 0$ or $a_3 = a_4 = 0$ or $a_5 = 0$ or $a_6 = 0$).

`Type2_2`. K/k should be a quadratic extension. $T(k) = K^*$ and T acts on $\mathbf{P}^6$ to give an X with degree 4 minimal parametrization. *pt* satisfies not($a_1 = a_2 = 0$ or $a_3 = a_4 = 0$ or $a_5 = a_6 = 0$).

`Type2_3`. K/k should be a quadratic extension. $T(k) = K^{*N_{K/k}=1} \times K^{*N_{K/k}=1}$. *pt* satisfies not($a_1 = a_2 = 0$ or $a_3 = a_4 = 0$ or $a_5 = a_6 = 0$).

`Type3`. K/k should be a cubic extension. $T(k) = K^{*N_{K/k}=1}$. *pt* satisfies not($a_1 = a_2 = a_3 = 0$ or $a_4 = a_5 = a_6 = 0$).

`Type4`. K/k and $K1/k$ should be distinct quadratic extensions. $T(k) = L^{*N_{L/K}=1}$ where L is $K.K1$. *pt* satisfies not($a_1 = a_2 = a_3 = a_4 = 0$ or $a_5 = a_6 = 0$).

`Type6`. K/k should be a degree 6 extension which contains cubic and quadratic subextensions K_3 and K_2. For simplicity, the precise condition is that the generator $y = K.1$ must have minimal polynomial of the form $x^6 + 2ax^4 + a^2x^2 - d$ and then $K_3 = k(y^2)$ and $K_2 = k(y^3 + ay)$. $T(k) = K^{*N_{K/K_3}=N_{K/K_2}=1}$. *pt* satisfies not($a_1 = a_2 = a_3 = a_4 = a_5 = a_6 = 0$).

`ParametrizeDelPezzoDeg6(X)`

This variant for parametrizing a degree 6 Del Pezzo also handles the degenerate (singular) case. Note however, that it doesn't recognise singularity from the Lie algebra computation as occurs for degrees 7 and 8. It tests for singularity at the start using the generic non-singularity computation that can be very slow. Therefore for known non-degenerate Del Pezzos of degree 6, it is always better to use the above `ParametrizeDegree6DelPezzo` directly.

That is also used here, if X turns out to be non-singular. Otherwise, projection from a singular point to $\mathbf{P}^5$ reduces the problem to that of parametrizing a rational scroll.

Example H116E20

In the this example, we start with a degree 3 Del Pezzo surface - a non-singular hypersurface in $\mathbf{P}^3$ - which contains the 3 disjoint lines $x = y = 0$, $z = t = 0$ and $x = z, y = t$. These are blown down to give a degree 6 Del Pezzo surface, the parametrisation of which gives a parametrisation of the original surface. As well as demonstrating the degree 6 code, this is a nice example of blowing down exceptional lines on surfaces, something for which more general code will be added at a future date.

```
> R3<x,y,z,t> := PolynomialRing(Rationals(),4,"grevlex");
> P3 := Proj(R3);
> //equation of the degree 3 surface:
> F := -x^2*z + x*z^2 - y*z^2 + x^2*t - y^2*t - y*z*t + x*t^2 + y*t^2;
> X3 := Scheme(P3,F);
> // get the ideal defing the union of the 3 lines:
> I1 := ideal<R3|[x,y]>;
> I2 := ideal<R3|[z,t]>;
> I3 := ideal<R3|[x-z,y-t]>;
> I := I1*I2*I3;
> I := Saturation(I);
```

General surface theory tells us that if H is the hyperplane divisor on X_3, then the blowing down is given by the projective map associated to the divisor $H + L_1 + L_2 + L_3$, where the L_i are our 3 lines. We need the global sections of the sheaf of this: if $L_1 + L_2 + L_3 \sim 2H - D$ (linear equivalence of divisors) for an effective divisor D, then the space of global sections“ is” the degree 3 graded part of the ideal of D (mod the equation of X_3). The ideal I_D of a suitable D is computed by requiring that $ID \cap I = (F, F_2)$ with F_2 a degree 2 polynomial in I.

```
> F2 := Basis(I)[5]; F2;
y*z - x*t
> ID := ColonIdeal(ideal<R3|[F,F2]>,I);
> ideal<R3|[F,F2]> eq (ID meet I);
true
> // get basis of degree 3 graded part of ID
> ID3 := ID meet ideal<R3|Setseq(MonomialsOfDegree(R3,3))>;
> B3 := MinimalBasis(ID3);
> B3;
[
    y*z*t - x*t^2,
    z^3 - z^2*t + t^3,
    y*z^2 - x*z*t,
    x*z^2 - x*z*t + y*t^2,
    y^2*z - x*y*t,
    x*y*z - x^2*t,
    x^2*z - x^2*t + y^2*t,
    x^3 - x^2*y + y^3
]
> // and a complementary subspace of F
> F in ideal<R3|Remove(B3,7)>;
false
> B3 := Remove(B3,7);
> // now map to the degree 6 Del Pezzo
> P6<a,b,c,d,e,f,g> := ProjectiveSpace(Rationals(),6);
> blow_down := map<X3->P6|B3>;
> X6 := blow_down(X3);
> Dimension(X6); Degree(X6);
2
6
```

We also need the inverse of blow down. The general `IsInvertible` function could be used here but again the general theory tells us that the inverse is given by linear equations and it is faster to find them directly by a Grobner basis plus linear algebra computation. We omit this for brevity and just assume the result.

```
> X3toX6 := iso<X3->X6|B3,[f,e,c,a]>;
> // now parametrise X6
> boo,prm := ParametrizeDegree6DelPezzo(X6);
> boo;
true
> p2toX3 := Expand(prm*Inverse(X3toX6));
> p2toX3;
Mapping from: Projective Space of dimension 2
Variables : $.1, $.2, $.3 to Sch: X3
with equations :
```

```
-77/9*$.1^3 + 59/6*$.1^2*$.2 + 10/3*$.1^2*$.3 + 8/9*$.1*$.2^2 -
    73/18*$.1*$.2*$.3 - 113/18*$.1*$.3^2 - 59/9*$.2^3 + 383/18*$.2^2*$.3 -
    259/9*$.2*$.3^2 + 329/18*$.3^3
253/18*$.1^3 - 193/6*$.1^2*$.2 - 17/3*$.1^2*$.3 + 695/18*$.1*$.2^2 -
    244/9*$.1*$.2*$.3 + 353/18*$.1*$.3^2 - 151/9*$.2^3 + 185/9*$.2^2*$.3 -
    41/9*$.2*$.3^2 - 79/9*$.3^3
-11/6*$.1^3 + 37/6*$.1^2*$.2 + 10/3*$.1^2*$.3 - 28/3*$.1*$.2^2 + 4*$.1*$.2*$.3 -
    7*$.1*$.3^2 + 8/3*$.2^3 + 8/3*$.2^2*$.3 - 11/2*$.2*$.3^2 + 9/2*$.3^3
11/18*$.1^3 + 8/3*$.1^2*$.2 - 1/6*$.1^2*$.3 - 28/9*$.1*$.2^2 - 2/9*$.1*$.2*$.3 -
    59/18*$.1*$.3^2 - 2/9*$.2^3 + 34/9*$.2^2*$.3 - 53/18*$.2*$.3^2 + 53/18*$.3^3
and inverse
-884/23043*x^3 + 884/23043*x^2*y - 884/23043*y^3 - 4436/23043*x*y*z -
    4334/23043*y^2*z + 6902/23043*x*z^2 - 3560/7681*y*z^2 - 4420/23043*z^3 +
    4436/23043*x^2*t + 4334/23043*x*y*t + 3778/23043*x*z*t + 4420/23043*y*z*t +
    4420/23043*z^2*t - 4420/23043*x*t^2 + 6902/23043*y*t^2 - 4420/23043*t^3
-442/23043*x^3 + 442/23043*x^2*y - 442/23043*y^3 - 3544/23043*x*y*z -
    6808/23043*y^2*z + 8800/23043*x*z^2 - 4392/7681*y*z^2 - 6290/23043*z^3 +
    3544/23043*x^2*t + 6808/23043*x*y*t + 4376/23043*x*z*t + 8744/23043*y*z*t +
    6290/23043*z^2*t - 8744/23043*x*t^2 + 8800/23043*y*t^2 - 6290/23043*t^3
-884/23043*x^3 + 884/23043*x^2*y - 884/23043*y^3 - 458/23043*x*y*z -
    5660/23043*y^2*z + 5828/23043*x*z^2 - 2854/7681*y*z^2 - 3910/23043*z^3 +
    458/23043*x^2*t + 5660/23043*x*y*t + 2734/23043*x*z*t + 6208/23043*y*z*t +
    3910/23043*z^2*t - 6208/23043*x*t^2 + 5828/23043*y*t^2 - 3910/23043*t^3
and alternative inverse equations:
...
```

```
ParametrizeDegree5DelPezzo(X)
```

Let X be a degree 5 Del Pezzo surface anticanonically embedded in 5-dimensional projective space over a number field. We allow that X can be a degenerate (singular) Del Pezzo here. The scheme X is always parametrizable over the base field and this intrinsic returns such a parametrisation without reduction to higher degree.

The scheme X has a finite automorphism group in this case, so the Lie Algebra method cannot be applied. However, there is a more geometric method using projections that works well for degree 5 and that is used here.

```
ParametrizeSingularDegree3DelPezzo(X,P2)
```

```
ParametrizeSingularDegree4DelPezzo(X,P2)
```

These two intrinsics compute whether a degree 3 (resp. 4) anticanonically embedded *singular* Del Pezzo X has a parametrization over the base number field k and, if so, return such a parametrization as a scheme map with inverse from $P2$ to X. A projective plane $P2$ over the same base field k is the second argument of the intrinsic and will be used as the domain of the map returned.

The conditions on X mean that is an irreducible degree 3 hypersurface in P^3 in the first case or an irreducible complete intersection of 2 quadrics in P^4 in the second case, having only a finite number of singularities that are canonical A-D-E type in either case. The condition that there is a finite non-empty set of singularities is checked but whether these singularities are canonical is not checked. In the unlikely event that a degree 3 hypersurface or degree 4 complete intersection has finitely many singularities but one is non-canonical, the functions will fail at some point.

If there is a singular point p defined over the base field, projection from p gives an immediate inverse parametrization of X in the degree 3 case and maps X onto a line or conic bundle in P^3 in the degree 4 case, which is then parameterized by the special routines for those cases. There remain a small number of configurations of conjugate singularities in the contrary case, corresponding to certain special root subsytems of E_6 or D_5. For these, individual methods have been devised and implemented. These include an adaptation of the Lie algebra method for the degree 3 and 4 singular Del Pezzo surfaces that are actually toric.

Example H116E21

The following is an example of a degree 3 hypersurface in P^3 over Q, that is a singular Del Pezzo with 4 conjugate A_1 singularities. It is handled very easily by the special case code.

```
> Q := RationalField();
> P2<a,b,c> := ProjectiveSpace(Q,2);
> P3<x,y,z,t> := ProjectiveSpace(Q,3);
> X := Scheme(P3, -4*x^2*y + 16*x*y^2 - y^3 + 2*x^2*z - 2*x*y*z +
>     7/2*y^2*z - 252*x*z^2 + 16*y*z^2 - 55*z^3 + 10*x^2*t + 14*x*y*t -
>     61/2*y^2*t - 3400*x*z*t + 216*y*z*t - 261*z^2*t - 11468*x*t^2 +
>     728*y*t^2 + 3987*z*t^2 + 21889*t^3);
> ParametrizeSingularDegree3DelPezzo(X,P2);
true Mapping from: Prj: P2 to Sch: X
with equations :
1/4*a^3 + 435/2*a^2*b - 4743/4*a*b^2 + 968*b^3 + 257/16*a^2*c + 183/8*a*b*c -
    3647/16*b^2*c + 8*a*c^2 - 8*b*c^2 + c^3
-257/32*a^3 + 2547/32*a^2*b - 10419/32*a*b^2 + 8129/32*b^3 - 4*a^2*c + 22*a*b*c
    - 66*b^2*c - 1/2*a*c^2 + 1/2*b*c^2
-57/32*a^3 - 173*a^2*b + 26459/32*a*b^2 - 1529/16*b^3 - 29/2*a^2*c + 83/4*a*b*c
    - 25/4*b^2*c - 7/2*a*c^2 + b*c^2
-1/32*a^3 + 457/16*a^2*b - 4081/32*a*b^2 + 33/2*b^3 + 2*a^2*c - 7/4*a*b*c -
    1/4*b^2*c + 1/2*a*c^2
and inverse
x*y*z - 2*y^2*z + 63/2*z^3 + 2*x^2*t - 9*x*y*t + 35/2*y^2*t + 299/2*z^2*t -
    4567/2*z*t^2 - 25075/2*t^3
x^2*z - 1/4*y^2*z + 4*z^3 + 7*x^2*t - 2*x*y*t + 9/4*y^2*t + 19*z^2*t - 290*z*t^2
    - 1593*t^3
x^3 - 63/4*x*y^2 + y^3 + 256*x*z^2 - 65/4*y*z^2 + 3454*x*z*t - 439/2*y*z*t +
    11650*x*t^2 - 2957/4*y*t^2
```

116.4.4 Minimization and Reduction of Surfaces

Given an algebraic variety defined by several polynomials with integer coefficients, *reduction* asks for another embedding of this **Z**-scheme, such that the defining polynomials have smaller coefficients. *Minimization* asks for an isomorphic **Q**-scheme with minimal invariants. Many constructions of algebraic varieties lead to very bad models and thus it becomes necessary to perform minimization and reduction. Otherwise, subsequent calculations become impractical. The result of the minimization process is usually not unique.

Minimization is done locally for each prime of bad reduction. The local minimization routines and the reduction routines are directly accessible. They may be helpful for local computations or if the computation of all bad primes is too slow. Note that these subroutines do not check for semi-stability (in the sense of Mumford's geometric invariant theory). Unstable varieties may lead to infinite loops. As smooth hypersurfaces are known to be stable the initial computation of the bad primes will fail if an unstable variety is given.

In this section, minimization and reduction routines are described for Del Pezzo surfaces of degrees 3 (cubic surfaces) and 4.

Minimization and reduction is also available for various kinds of genus one curves (see Section 124.6) and plane quartics (see Section 114.12.3).

```
MinimizeCubicSurface(f, p)
```

Verbose `MinRedCubSurf` *Maximum* : 2

Given a cubic surface f as a homogeneous polynomial with integer coefficients, this routine performs a minimization at the place p. The new equation and the transformation matrix are returned. No checks of stability are done so that an unstable surface will lead to an infinite loop.

```
ReduceCubicSurface(f)
```

Verbose `MinRedCubSurf` *Maximum* : 2

Given a cubic surface f as a homogeneous polynomial with integral coefficients, this function computes a reduction of the surface. The second returned value is the transformation used.

```
MinimizeReduceCubicSurface(f)
```

Verbose `MinRedCubSurf` *Maximum* : 2

Given a smooth cubic surface f as a homogeneous polynomial with integer coefficients, this function computes a minimized and reduced model of the surface. The second return value is the transformation matrix. The transformation matrix applied to f will evaluate to a scalar multiple of the returned polynomial.

The algorithm is based on [Els].

`MinimizeDeg4delPezzo(f, p)`

Verbose `MinRedDeg4delPezzo` *Maximum :* 1

Given a degree 4 del Pezzo surface f as a sequence of two quadrics with integer coefficients, this function will compute a partially local minimized model for the place p. The second return value is the transformation matrix.

`MinimizeReduceDeg4delPezzo(f)`

Verbose `MinRedDeg4delPezzo` *Maximum :* 1

Given a degree 4 del Pezzo surface as a sequence of two quadrics with integral coefficients, this function computes a practically minimized and reduced model of the surface. The second return value is the transformation matrix.

The transformation matrix applied to the initial polynomials will evaluate to polynomials defining the same **Q** scheme as that defined by the returned quadrics.

For the reduction step, `ReduceQuadrics` is called.

`MinimizeReduce(S)`

Verbose `MinRedCubSurf` *Maximum :* 2

Verbose `MinRedDeg4delPezzo` *Maximum :* 1

Given a del Pezzo surface S of degree 3 or 4 this function will call the minimization and reduction routines described above and converts the output scheme X to a del Pezzo surface. The second returned value is the matrix hat maps S to the result.

Example H116E22

This example demonstrates minimization and reduction on del Pezzo surfaces of degree 3 and 4 obtained by blowing up rational points.

```
> P2 := ProjectiveSpace(RationalField(),2);
> pts := [P2| [-5,-10,-8], [-4,10,-4], [8,-2,-5], [0,-10,0], [1,5,7], [-7,-8,-6]];
> S := DelPezzoSurface(pts);
> _<W, X, Y, Z> := AmbientSpace(S); // give names to the variables
> S;
Del Pezzo Surface of degree 3 over Rational Field defined by
-W*X^2 + 318827/104630*X^3 + 46774615/29003436*W*X*Y -
    2039633371/290034360*X^2*Y - 2588798/7250859*W*Y^2 +
    246700427/58006872*X*Y^2 - 4904503/7250859*Y^3 + W^2*Z -
    318827/104630*W*X*Z + 34829/52315*X^2*Z + 117476057/58006872*W*Y*Z -
    2004449/27622320*X*Y*Z + 4769241/12890416*Y^2*Z - 34829/52315*W*Z^2 -
    44696243/72508590*Y*Z^2
> MinimizeReduce(S);
Del Pezzo Surface of degree 3 over Rational Field defined by
-22*W^2*X + 38*W*X^2 - 4*X^3 - 28*W^2*Y + 103*W*X*Y - 48*X^2*Y + 44*W*Y^2 -
    59*X*Y^2 - 24*Y^3 + 24*W^2*Z + 80*W*X*Z + 44*X^2*Z - 57*W*Y*Z + 73*X*Y*Z
    - 59*Y^2*Z - 79*W*Z^2 - 21*X*Z^2 - 5*Y*Z^2
```

Now we consider the surface of degree 4 obtained by blowing up only the first five points.

```
> T := DelPezzoSurface(pts[1..5]);
> _<V, W, X, Y, Z> := AmbientSpace(T);
> T;
Del Pezzo Surface of degree 4 over Rational Field defined by
-W^2 + 7031/194432*X^2 + V*Y + 10877/13888*X*Y + 47/280*W*Z + 412801/277760*X*Z
     - 78153/138880*Y*Z - 86217/9721600*Z^2,
-W*X + 1693/6944*X^2 + 23/496*X*Y + V*Z + 26763/9920*X*Z - 4003/4960*Y*Z
     - 233411/347200*Z^2
> MinimizeReduce(T);
Del Pezzo Surface of degree 4 over Rational Field defined by
-5*V^2 + 4*V*W + 8*W^2 + V*X - 8*X^2 - 8*V*Y + 3*W*Y + 3*X*Y + 15*Y^2 -
    2*V*Z - 16*X*Z - 10*Y*Z,
-2*V^2 + V*W + 3*W^2 + 2*V*X + 2*W*X - 6*X^2 + 3*V*Y + 3*W*Y - 17*X*Y -
    5*Y^2 + 5*V*Z - 22*X*Z - Y*Z + 13*Z^2
```

116.4.5 Cubic Surfaces over Finite Fields

In this section all cubic surface are represented by a homogeneous polynomial of degree 3 in a rank 4 polynomial ring. The coefficients are elements of a finite field.

```
NumberOfPointsOnCubicSurface(f)
```

Given a smooth cubic surface f over a finite field this routine computes the Frobenius action on the lines. The return values are the number of points of the surface and the Swinnerton-Dyer number of conjugacy class of the Weil group $W(E_6)$ that contains the Frobenius.

Example H116E23

```
> p := NextPrime(3^100);
> r<x,y,z,w> := PolynomialRing(GF(p),4);
> S := x^3 + 2* y^3 + 7* z^3 + 11 * w^3 - 5 * (-x-y-z-w)^3;
> NumberOfPointsOnCubicSurface(S);
2656139888758747693387813220357796268292334528059462421112503157258849853119260\
    79714208525578202
13
```

So we get a large number of points and the Frobenius has Swinnerton-Dyer number 13.

```
IsIsomorphicCubicSurface(f,g)
```

UseLines BoolElt *Default :* false

Given cubic surfaces f and g defined over finite fields, the intrinsic returns `true` if the surfaces are isomorphic. If f and G are isomorphic, there is a second return value comprising a list of matrices such that g^m will evaluate to a scalar multiple of f for each matrix m in the list. In the case where there are several isomorphisms over the algebraic closure of the basefield, one matrix for each isomorphism is returned.

Note that an isomorphism of smooth cubic surfaces is always given by a linear map.

The computation is based on an analysis of a finite set of points associated to the surface. Here we used the singularities of the hessian. If the hessian degenerates, the 135 intersection points of the lines are used. Setting `UseLines` to `true` indicates that the second algorithm is to be used.

As the algorithm involves huge field extensions it is only practical for surfaces over finite fields.

Example H116E24

```
> _<x,y,z,w> := PolynomialRing(GF(101),4);
> S := x^3 + y^3 + z^3 + w^3 - (x+y+z+w)^3;
> time a,b := IsIsomorphicCubicSurface(S, S);
Time: 0.530
> #b;
120
> S := x^3 + 2*y^3 + 7*z^3 + 5*w^3 - y*z*w + x^2*w + 2*y*z^2;
> time a,b := IsIsomorphicCubicSurface(S, S);
Time: 0.480
> #b;
1
> S := x^3 + y^3 + z^3 + w^3;
> time a,b := IsIsomorphicCubicSurface(S, S);
Time: 22.830
> #b;
648
```

Thus the diagonal cubic surface has 648 automorphisms and the Clebsch cubic surface has 120. Both examples are exceptional, a general cubic surface has a trivial automorphism group. The first example is much slower because the hessian degenerates and the 27 lines are used.

116.4.6 Construction of Cubic Surfaces

```
CubicSurfaceByHexahedralCoefficients(pol)
```

Given a separable polynomial p of degree 6 this intrinsic constructs a cubic surface having the roots of the p as hexahedral coefficients. These surfaces automatically have a Galois invariant set of 12 lines.

See [EJ10] for details.

```
CoblesRadicand(p)
```

Given a separable polynomial p of degree 6 this routine evaluates the Cobles quartic at the roots of p. Up to a square factor this value is the discriminant of the cubic surface constructed using hexahedral coefficients.

Example H116E25

```
> q<tt> := PolynomialRing(RationalField());
> p6 := tt^6 + 34*tt^4 + 180*tt^3 + 458*tt^2 + 524*tt + 212;
> CoblesRadicand(p6);
-676
> eqn := CubicSurfaceByHexahedralCoefficients(p6);
> Max([AbsoluteValue(c) : c in Coefficients(eqn)]);
1302161870313141409337256000 20
```

We have to use minimization and reduction to make further computations faster.

```
> S := MinimizeReduce(DelPezzoSurface(eqn));
> Equation(S);
6*y[1]^3 - 14*y[1]^2*y[2] + 6*y[1]*y[2]^2 - 6*y[2]^3 - 14*y[1]^2*y[3] +
    9*y[1]*y[2]*y[3] + 11*y[2]^2*y[3] - 21*y[1]*y[3]^2 + 14*y[2]*y[3]^2 +
    3*y[3]^3 - 3*y[1]*y[2]*y[4] + 10*y[2]^2*y[4] + 8*y[1]*y[3]*y[4] -
    53*y[2]*y[3]*y[4] + 40*y[3]^2*y[4] + 9*y[1]*y[4]^2 + 39*y[2]*y[4]^2 -
    23*y[3]*y[4]^2 - 16*y[4]^3
> M := PicardGaloisModule(S);
> Order(Group(M));
72
> CohomologyGroup( CohomologyModule(Group(M),M), 1);
Full Quotient RSpace of degree 2 over Integer Ring
Column moduli:
[ 2, 2 ]
```

Here the hexahedral approach gives us a cubic surface with nontrivial cohomology.

116.4.7 Invariant Theory of Cubic Surfaces

For background on this classical topic we refer to [Hun96, Appendix B] and [Sal58].

In this section a cubic surface is represented by a homogeneous polynomial of degree 3 in a rank 4 polynomial ring.

116.4.7.1 Invariants

By a theorem of Clebsch the ring of invariants of a cubic surface is generated by 5 invariants having degrees 8, 16, 24, 32 and 40. An explicit system of generators was found by Salmon. By Geometric Invariant Theory, stable cubic surfaces are isomorphic if and only if their invariants determine the same point in the weighted projective space $\mathbf{P}(1, 2, 3, 4, 5)$.

```
ClebschSalmonInvariants(f)
```

Computes a sequence of the numerical values of Salmon's invariants of the cubic surface given by the polynomial f. The second returned value is the discriminant of the surface.

```
SkewInvariant100(f)
```

Computes the numerical value a degree 100 skew invariant I_{100} of the cubic surface f. The square of I_{100} is an element of Clebsch invariant ring. It vanishes if and only if the cubic surface has an Eckardt point.

```
CubicSurfaceFromClebschSalmon(inv)
```

Computes a cubic surface whose invariants are equal to the given sequence. The algorithm requires the last invariant to be non-zero.

Example H116E26

```
> r4<x,y,z,w> := PolynomialRing(Rationals(),4);
> surf := r4!CubicSurfaceFromClebschSalmon([1,2,3,4,5]);
> surf := r4!MinimizeReduceCubicSurface(surf);
> surf;
-79*x^3 - 64*x^2*y + 228*x^2*z - 197*x^2*w + 320*x*y^2 - 470*x*y*z + 492*x*y*w
  - 180*x*z^2 - 94*x*z*w - 242*x*w^2 - 125*y^3 + 100*y^2*z + 94*y^2*w + 530*y*z^2
  - 886*y*z*w + 390*y*w^2 - 235*z^3 + 526*z^2*w - 825*z*w^2 +  279*w^3
> inv := ClebschSalmonInvariants(surf);
> inv;
[ 976235771549603375/3, 1906072563306098780753436239622781250/9,
930388109734329783009461918136480101451041525943359375/9,
3633112616588281944217451032208493848831995668840667046827293985351562500/81,
44334681229770747115131288025953470580778533302801673727424367422761364246\
35758514404296875/243 ]
> [inv[i] / inv[1]^i : i in [1..5]];
[ 1, 2, 3, 4, 5 ]
> SkewInvariant100(surf);
-1474765875168770247752210363977205595498018662672331422683943150206680481\
86921012313818705064245354658498892851426776914304662591690689504117661282\
58105510099234779123372779972973056799912669452615967667952645570039749145\
50781250/531441
```

Thus the constructed surface has equivalent invariants (as they are in $\mathbf{P}(1, 2, 3, 4, 5)$) and no Eckardt points.

116.4.7.2 Covariants

A covariant is in the same ambient space as the initial surface. For example the equation itself is a covariant. In the case of a cubic surface this gives a degree 1 order 3 covariant. Products of covariants are again covariants. They form a ring over the ring of invariants.

```
LinearCovariants(f)
```

The intrinsic constructs a sequence containing Salmon's 4 linear covariants for the cubic surface f.

```
ClassicalCovariantsOfCubicSurface(f)
```

The intrinsic constructs a sequence containing the 4 classical covariants of the cubic surface f. The first one is the hessian. The next two are classically known as T and Θ. The last one is a degree 9 surface, which intersects f precisely in its 27 lines.

116.4.7.3 Contravariants

A contravariant is in the dual projective space of the initial surface. All contravariants form a ring over the ring of invariants. Contravariants can be constructed from invariants of varieties of the same degree but dimension one less by the Clebsch transfer principle.

```
NumericClebschTransfer(f, inv, p)
```

Given a form f and a user program `inv` that evaluates an invariant of a form of the same degree but one variable less, this function evaluates the corresponding contravariant of f at the point p. If this is repeated using sufficiently many different knots, it is possible to reconstruct a polynomial representation of the contravariant by interpolation.

```
ContravariantsOfCubicSurface(f)
```

Computes a sequence of 3 contravariants of the cubic surface f. By Clebsch transfer they correspond to the invariants S, T, and the discriminant of plane cubic curves. Thus the first one describes all hyperplanes such that the intersection with $f = 0$ gives a cubic curve with j-invariant equal to zero. The second gives all hyperplanes intersecting $f = 0$ in a cubic curve with j-invariant 1728 (as long as the intersection is smooth). The last one is $S^2 - 6T$. This is the degree 12 polynomial of the (formal) dual surface. It describes all hyperplanes such that the intersection with $f = 0$ is singular. For smooth surfaces this is equivalent to tangency. If $f = 0$ is singular the result will be reducible or even zero.

Example H116E27

This is the Cayley cubic surface. It has 4 singularities of type A_1 and each results in a linear factor of multiplicity two in the (formal) dual surface.

```
> r4<x,y,z,w> := PolynomialRing(Rationals(),4);
> surf := x*y*z + x*y*w + x*z*w + y*z*w;
> cont := ContravariantsOfCubicSurface(surf);
> Factorization(cont[3]);
```

```
[
    <w, 2>,
    <z, 2>,
    <y, 2>,
    <x, 2>,
    <x^4 - 4*x^3*y - 4*x^3*z - 4*x^3*w + 6*x^2*y^2 + 4*x^2*y*z + 4*x^2*y*w
     + 6*x^2*z^2 + 4*x^2*z*w + 6*x^2*w^2 - 4*x*y^3 + 4*x*y^2*z + 4*x*y^2*w
     + 4*x*y*z^2 - 40*x*y*z*w + 4*x*y*w^2 - 4*x*z^3 + 4*x*z^2*w + 4*x*z*w^2
     - 4*x*w^3 + y^4 - 4*y^3*z - 4*y^3*w + 6*y^2*z^2 + 4*y^2*z*w + 6*y^2*w^2
     - 4*y*z^3 + 4*y*z^2*w + 4*y*z*w^2 - 4*y*w^3 + z^4 - 4*z^3*w + 6*z^2*w^2
     - 4*z*w^3 + w^4, 1>
]
```

116.4.7.4 Interaction of Covariants and Contravariants

One can apply a contravariant to a covariant (or vice versa). The result is a new covariant (resp. contravariant) or an invariant. Its degree is the sum of the degrees of the arguments. The order is the difference of the two orders. If the order of the result is zero, it is an invariant.

One way to define the action is to interpret the contravariant as a differential operator, i.e. x_i^k acts as $\frac{\partial^k}{\partial x_i^k}$. Then one applies this differential operator to the covariant.

```
ApplyContravariant(c, d)
```

Given a covariant c and a polynomial d, this intrinsic interprets d as a differential operator (i.e., x is replaced by d/dx). It applies this operator to the polynomial c and returns the resulting polynomial.

In invariant theory d is a contravariant and c is a covariant.

Example H116E28

Here we compute Salmon's first invariant by applying a degree 4 order 4 contravariant to the hessian which is a degree 4 order 4 covariant.

```
> r4<x,y,z,w> := PolynomialRing(RationalField(),4);
> surf := x^3 + 2*y^3 + 3*z^3 + 5*w^3 - 2*x*y*(z-w) + (x+y+z+w)^3;
> cont := ContravariantsOfCubicSurface(surf);
> cov := ClassicalCovariantsOfCubicSurface(surf);
> ApplyContravariant(cont[1],cov[1]) / (2^11 * 3^9);
1438753/729
> ClebschSalmonInvariants(surf)[1];
1438753/729
```

116.4.8 The Pentahedron of a Cubic Surface

A general cubic surface can be written as a sum of 5 cubes of linear forms. These are unique up to scaling by third roots of unity and permutation. Thus we can associate 5 points in the dual projective space to a given cubic surface. They are called the faces of its pentahedron. In general the faces of the pentahedron are defined over a larger field.

The algorithm is described in [RS00].

```
PentahedronIdeal(f)
```

Computes the ideal of the faces of the pentahedron of the cubic surface f.

Example H116E29

The first example is a randomly chosen cubic surface. By construction it has a proper rational pentahedron.

```
> r4<x,y,z,w> := PolynomialRing(Rationals(),4);
> surf := x^3 + (x-y+2*z)^3 + (y-w)^3 + z^3 + (x - 3*y-2*z-7*w)^3;
> p_id := PentahedronIdeal(surf);
> Points(Cluster(ProjectiveSpace(Rationals(),3),Basis(p_id)));
{@ (-1/7 : 3/7 : 2/7 : 1), (0 : -1 : 0 : 1), (0 : 0 : 1 : 0),
   (1/2 : -1/2 : 1 : 0), (1 : 0 : 0 : 0) @}
```

The next example shows that the pentahedron of the diagonal cubic surface degenerates.

```
> diag := x^3 + y^3 + z^3 + w^3;
> p_id2 := PentahedronIdeal(diag);
> Points(Cluster(ProjectiveSpace(Rationals(),3),Basis(p_id2)));
{@ (0 : 0 : 0 : 1), (0 : 0 : 1 : 0), (0 : 1 : 0 : 0), (1 : 0 : 0 : 0) @}
```

The final example is a surface without a pentahedron.

```
> degen := x^3+y^3+z^3 + x*y*z+ w^3;
> p_id3 := PentahedronIdeal(degen);
> Points(Cluster(ProjectiveSpace(Rationals(),3),Basis(p_id3)));
{@ (0 : 0 : 0 : 1) @}
```

116.5 Bibliography

[**Bec07**] Tobias Beck. Formal Desingularization of Surfaces – The Jung Method Revisited –. Technical Report 2007-31, RICAM, December 2007. URL:http://www.ricam.oeaw.ac.at/publications/reports/.

[**Bec08**] Tobias Beck. Software Documentation. Technical Report 2008-8, RICAM, May 2008. URL:http://www.ricam.oeaw.ac.at/publications/reports/.

[**BHPdV04**] Barth, Hulek, Peters, and Van de Ven. *Compact Complex Surfaces.* Ergebnisse der Mathematik und ihrer Grenzgebiete 4. Springer, second edition, 2004.

[**BS08**] Tobias Beck and Josef Schicho. Adjoint Computation for Hypersurfaces Using Formal Desingularizations. Technical Report 2008-2, RICAM, January 2008. URL:http://www.ricam.oeaw.ac.at/publications/reports/.

[**DES93**] Decker, Ein, and Schreyer. Construction of surfaces in P4. *J. Algebraic Geometry*, 2:185–237, 1993.

[**dG06**] W. de Graaf, M. Harrison, J. Pilnikova and J. Schicho. A Lie Algebra Method for Rational Parametrization of Severi-Brauer Surfaces. *J. Alg.*, 303(2):514–529, 2006.

[**dGP**] W.A. de Graaf and J. Pilnikova. Parametrizing Del Pezzo surfaces of degree 8 using Lie algebras. URL:http://arxiv.org/abs/math.NT/0512477.

[**EJ10**] Andreas-Stephan Elsenhans and Jörg Jahnel. Cubic surfaces with a Galois invariant double-six. *Cent. Eur. J. Math.*, 8(4):646–661, 2010.

[**Els**] Andreas-Stephan Elsenhans. Good models for cubic surfaces. (To appear).

[**GSHPBS12**] J. Gonzalez-Sanchez, M. Harrison, I. Polo-Blanco, and J. Schicho. Algorithms For Del Pezzo Surfaces of Degree 5 (Construction, Parametrization). 2012.

[**Hor76**] E. Horikawa. Algebraic Surfaces of General Type with Small c_1^2. II. *Invent. Math.*, 37:121–155, 1976.

[**HS06**] M.C. Harrison and J. Schicho. Rational Parametrisation for Degree 6 Del Pezzo Surfaces using Lie Algebras. In *Proceedings ISSAC'06*, 2006.

[**Hun96**] Bruce Hunt. *The geometry of some special arithmetic quotients*, volume 1637 of *Lecture Notes in Mathematics.* Springer-Verlag, Berlin, 1996.

[**Man86**] Yu. I. Manin. *Cubic Forms (2nd ed.).* North-Holland Publishing Co., Amsterdam, 1986. Vol. 4 of North-Holland Mathematical Library.

[**RS00**] Kristian Ranestad and Frank-Olaf Schreyer. Varieties of sums of powers. *J. Reine Angew. Math.*, 525:147–181, 2000.

[**Sal58**] George Salmon. *A treatise on the analytic geometry of three dimensions.* Revised by R. A. P. Rogers. 7th ed. Vol. 1. Edited by C. H. Rowe. Chelsea Publishing Company, New York, 1958.

[**Sch98**] Josef Schicho. Rational parametrization of surfaces. *J. Symbolic Comput.*, 26(1):1–29, 1998.

[**Sch00**] Josef Schicho. Proper parametrization of surfaces with a rational pencil. In *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation (St. Andrews)*, pages 292–300 (electronic), New York, 2000. ACM.

[**SvdV87**] A.J. Sommese and A. van der Ven. On the adjunction mapping. *Math. Ann.*, 278:593–603, 1987.

117 HILBERT SERIES OF POLARISED VARIETIES

Chapter 117
HILBERT SERIES OF POLARISED VARIETIES

117.1 Introduction

This chapter describes methods of generating graded rings that correspond to polarised algebraic varieties of various types. They can be used to generate examples of subcanonical curves—curves X polarised by a divisor D for which $kD = K_X$, the canonical class, for some integer k—K3 surfaces, Fano 3-folds and Calabi–Yau 3-folds. Of these, K3 surfaces are the best developed, and a database containing several thousand surfaces forms part of Magma.

117.1.1 Key Warning and Disclaimer

It is important to be aware of the nature and limitations of the output of the functions and databases described in this chapter. We list five of the issues in numbered points below, of which number 5 is the most important.

A typical example of a graded ring arises from a hyperelliptic curve C of genus g embedded in weighted projective space (wps):

$$C : (y^2 = f) \subset \mathbf{P}(1, 1, g + 1)$$

where the coordinates on $\mathbf{P}(1, 1, g + 1)$ are x_1, x_2, y of weights $1, 1, g + 1$ respectively, and $f = f_{2g+2}(x_1, x_2)$ is a homogeneous polynomial of degree $2g + 2$ in two variables having distinct roots. This embedded variety has homogeneous coordinate ring

$$R(C) = \frac{k[x_1, x_2, y]}{(y^2 - f)}$$

where k is the ground field. As a concise representation of this data, we record only a particular rational representation of the Hilbert series $P_R(t)$ of $R = R(C)$,

$$P_R(t) = 1 + 2t + 3t^2 + \cdots + (g + 1)t^g + (g + 3)t^{g+1} + \cdots = \frac{1 - t^{2g+2}}{(1 - t)(1 - t)(1 - t^{g+1})}$$

from which, as shown in Section 117.2.2, we deduce the fields

Weights = [1, 1, g+1], EquationDegrees = [2g+2].

Even though using this representation involves losing much information about the curve C, it has preserved enough detail so that it is still possible to do such things as create another curve having the same basic invariants as C, or to recognise the family of hypersurfaces of degree 2 in $\mathbf{P}(1,1,g+1)$ to which C belongs.

This example illustrates several of the points one should keep in mind when interpreting the output of most functions in this chapter.

1. Weighted projective space (wps) The methods used here automatically generate examples of varieties defined by (weighted) homogeneous equations in wps, and so one must be familiar with wps from the outset. See Fletcher [IF00] for an accessible introduction to wps if necessary.

2. Field of definition Since the functions described in this chapter do not return literal equations of varieties, they do not assign a base field k. It is useful to have in mind $k = \mathbf{C}$, the complex numbers, but in many cases the base field is not relevant and one could work over any field. Having said that, there are cases where the base field is a crucial part of the problem.

3. Polarised varieties and their graded rings A *polarised variety* X, A is a variety X together with a divisor A that is ample on X; that is, there is a multiple kA of A which is a hyperplane section of X in some projective embedding $X \subset \mathbf{P}^N$. The homogeneous coordinate ring of X in this embedding is a graded ring which is generated in degree 1. But this embedding is not necessarily the one we want: the graded ring may be very large. Instead, we consider the *total graded ring* of A

$$R(X, A) = \oplus_{n\geq 0} H^0(X, \mathcal{O}_X(nA)),$$

which, with very few exceptions, is a much smaller ring. We do not define the terms precisely here, but suffice it to say that the Proj-correspondence between varieties in wps and the graded rings that are their homogeneous coordinate rings holds in this context between X, A and $R(X, A)$ just as it does for embeddings in ordinary projective space.

Thus we regard the following three pieces of data as being equivalent:

- a polarised variety X, A;
- the total graded ring $R(X, A)$ of a polarised variety X, A;
- the embedding $X \subset \mathbf{P}^N(w_0, \ldots, w_N)$ by all multiples of A, for some weights $w_0, \ldots, w_N$.

And so we use the words ‘polarised variety’ and ‘graded ring’ interchangeably. The fact that we also use ‘variable’, ‘coordinate’ and ‘generator’ synonymously is another reflection of this equivalence.

4. Numerical data of families of varieties In fact, we do not consider a single polarised variety X, A. Instead we record weaker information that characterises a family of varieties of which X, A is a particular member. The key piece of information that we work with is the Hilbert series $P_R(t)$ of the graded ring $R = R(X, A)$. This is calculated using the Riemann–Roch formula (RR) once we have decided which class of varieties we are concerned with. In favourable cases, RR takes as ingredients some discrete pieces of geometric data such as genus (which are invariant in flat families of suitably prescribed varieties) and returns the dimension of the n-th graded piece R_n of the graded ring R. The algorithms work by taking such appropriate data as input, returning a Hilbert series (which is done by applying a formula that is hard-coded) and then analysing that series.

We can often produce extra information such as a prediction of weights $w_0, \ldots, w_N$ in which some suitable X is embedded. Some elementary examples of this are worked out Section 117.2.1.

5. Main problem In most cases, there are no criteria to determine whether a particular set of invariants for RR are actually the invariants of some polarised variety X, A. So even though the data that is then generated by Magma purports to be associated to some graded ring $R(X, A)$, there is no reason in any particular case why there really should exist such a polarised variety X, A. Fortunately, in many cases it is clear that there really is a variety that realises the output. In the example of the genus g hyperelliptic curve above, knowing the weights $(1, 1, g + 1)$ and the degree of the equation, it is easy to see that there exists a nonsingular variety with these data and one could even write Magma routines to present an example using the scheme machinery of Chapter 104.1.1 and then attempt to construct a Weierstrass model of the hyperelliptic curve as described in Chapter 104.1.1 to obtain access to the specialist machinery provided for such curves.

As a rule, any polarised variety X, A that is described by the output of a function—even by the K3 database—cannot be assumed to exist, or if it does exist, it might not take exactly the form described. To prove that such a variety exists as described, it is sufficient to show that there is a quasi-smooth variety in the given wps having the given Hilbert series (or Hilbert numerator).

117.1.2 Overview of the Chapter

Graded ring calculations can be carried out in many different contexts. Included here are functions that work with subcanonical curves, K3 surfaces, Fano 3-folds and Calabi–Yau 3-folds. The latter three have appeared recently as parts of PhD theses, by Altinok [Alt98], Suzuki [Suz] and Buckley [Buc03] respectively. Other references for some of this material are [ABR02], [Rei00], [Pap03], [Bro03].

Section 117.2 gives a sketch of the theory of Hilbert series and describes functions that compute Hilbert series from Hilbert polynomials. It includes worked out examples of the elementary calculations that are behind most of the chapter. It contains the important definition of *Hilbert numerator* with respect to a collection of weights, which turns out to be the key point when we try to describe graded rings as the coordinate rings of polarised varieties.

Singularities are a main ingredient of RR in many applications, and Section 117.3 describes their construction and properties.

Five of the next six sections are devoted to different classes of polarised varieties. In fact, there are four specific classes of polarised variety, and one general class which encompasses them. Section 117.4 contains functions that apply to the general class, and thus are inherited by all classes: when consulting later sections, one should bear in mind that most basic functions will be described in this section. Section 117.5 covers the first and most elementary application of graded ring methods to studying subcanonical curves, that is, curves polarised by a divisor that divides their canonical class.

K3 surfaces are described in Section 117.6. Although one can construct a single graded ring in isolation, a benefit of the graded ring methods is that they can be used generate large lists in one go. One such application is the amplification of Magma's K3 database from the 391 K3 surfaces in codimension at most 4 to the 24,099 cases in the current version. This is discussed in Section 117.7 which includes a precise statement characterising which K3 surfaces are included in the database and a further severe disclaimer.

There are two classes of 3-fold available: Fano 3-folds and Calabi–Yau 3-folds. The former is covered in Section 117.8, the latter in Section 117.9. Each of these is in a fairly early stage of development, having only basic creation functions and no systematic means for generating the large lists similar to those that exist for K3 surfaces. Nevertheless, one can begin to write lists. Section 117.10 describes how one can assemble such lists into Magma databases, although the process is somewhat technical and has its own limitations. Anyone attempting such lists will be aware that, following results of Kawamata, there are only finitely many deformation families of Fano 3-folds—Suzuki [Suz] classifies those of high Fano index—while it is still unknown whether or not there are finitely many families of Calabi–Yau 3-folds, Kreuzer and Skarke's vast lists [KS00] notwithstanding.

117.2 Hilbert Series and Graded Rings

The theory of Hilbert series is standard so it is only briefly touched on here. See Matsumura [Mat89] Section 13 for more details (or other any standard textbook on algebra such as Zariski–Samuel or Eisenbud.)

117.2.1 Hilbert Series and Hilbert Polynomials

Let $R = \oplus R_n$, the sum taken over $n \geq 0$, be a finitely-generated graded k-algebra with $R_0 = k$ and grading given by n. The *Hilbert polynomial* of R is the numerical polynomial $p(t)$ such that $\dim R_n = p(n)$ for all n sufficiently large. The *Hilbert series* of R is the power series

$$P(R) = \sum_{n \geq 0} (\dim R_n) t^n.$$

For example, let $C \subset \mathbf{P}^3$ be the twisted cubic defined by the three equations

$$xz = y^2, \quad xt = yz, \quad yt = z^2$$

and let

$$R = \frac{k[x,y,z,t]}{xz - y^2, xt - yz, yt - z^2}$$

be the homogeneous polynomial ring of C. Then one computes that

$$P(R) = 1 + 4t + 7t^2 + 10t^3 + \cdots$$

as follows. Certainly $R_0 = k$, and there are no equations of degree 1 so R_1 is the 4-dimensional vector space spanned by x, y, z, t. Now R_2 is spanned by the 10 quadrics $x^2, xy, \dots, t^2$, but these are related by the three equations, so $\dim R_2 = 10 - 3 = 7$. And so on. If you are systematic about it, you will discover quickly that the dimension of R_n is $3n + 1$ for $n \geq 1$. This can be proved by induction. So the Hilbert polynomial of R is $p(t) = 3t + 1$, which determines every coefficient of the Hilbert series $P(R)$ except the constant term.

Consider a bigger example: let $C = C_{10} \subset \mathbf{P}(1,1,5)$ be a hyperelliptic curve of genus 4 and degree 2 given by an equation

$$y^2 = f_{10}(x_1, x_2)$$

in coordinates x_1, x_2, y on wps $\mathbf{P}(1,1,5)$ where f_{10} is some general polynomial of degree 10. Again one calculates the Hilbert polynomial as $p(t) = 2t - 3$ and Hilbert series as

$$P(t) = 1 + 2t + 3t^2 + 4t^3 + 5t^4 + 7t^5 + 9t^6 + \cdots.$$

This power series can also be described as a rational function

$$P(t) = \frac{1 - t^{10}}{(1-t)^2(1-t^5)}$$

from which one can formulate a rule of thumb for relating this expression to the description of the curve as $C_{10} \subset \mathbf{P}(1,1,5)$ (or see the answer in Section 104.1.1 or [Rei00], Section 3.2).

Now there are other curves in wps that have the same Hilbert polynomial p as this example, but which have a different Hilbert series. To find one, consider modifying early terms of the series

$$P(t) = 1 + t + 2t^2 + 4t^3 + 5t^4 + 7t^5 + 9t^6 + \dots$$

and attempt to build the curve B in some wps. The coefficient of the term t (which is 1 in this case) implies that there is one linear coordinate x. Then the $2t^2$ demands one degree 2 coordinate y so that the 2-dimensional space of degree 2 homogeneous functions is spanned by x^2, y. In degree 3, x, y generate only a 2-dimensional space $\langle x^3, xy \rangle$, but the graded piece in degree 3 is 4-dimensional, according to the term $4t^3$, so there must be two more generators z_1, z_2. In degree 4 we see $x^4, x^2y, y^2, xz_1, xz_2$ which is just right if we assume that these are linearly independent (which we do). Similarly in degree 5 we hit the right number on the nose. But in degree 6 we see 11 monomials:

$$x^6, x^4y, x^2y^2, y^3, x^3z_1, xyz_1, x^3z_2, xyz_2, z_1^2, z_1z_2, z_2^2.$$

These lie in a 9-dimensional vector space (9 being the coefficient of t^6 in $P(t)$) so there must be two relations between them. We could carry on in this vein, but in fact we have done enough now: it is an exercise in wps to show that the curve

$$B = B_{6,6} \subset \mathbf{P}(1,2,3,3)$$

defined by two general polynomials of degree 6 has genus 4 and degree 2 just as in the first example. We will work this example in Magma in the next subsection.

Thus the Hilbert polynomial determines the coefficients of high powers of t in the Hilbert series. So in this case, and more generally for the subcanonical curves of Section 117.5, to determine a Hilbert series P one must produce both the Hilbert polynomial p and a sequence V of the early coefficients that are not determined by p.

There is one extra twist that can happen: it is possible that there is not a single Hilbert polynomial p that determines the higher coefficients, but a sequence of polynomials $p_0, \ldots, p_{r-1}$ so that the nth coefficient is given by $p_i(n)$ where i is the residue of n modulo r. It is easy to generate examples of this using the intrinsics described below. This phenomenon occurs naturally when the RR data includes quotient singularities or fractional divisors, since these tend to make periodic corrections to an otherwise well-determined Hilbert polynomial, and one could rephrase this as above by saying that there is a sequence of Hilbert polynomials. In practice, this is a small distraction that is completely bound up in the main creation intrinsics.

```
HilbertFunction(p,V)
```

```
HilbertFunction(Q,V)
```

The Hilbert function determined by a univariate polynomial p (the Hilbert polynomial), or a sequence Q of univariate polynomials, and a sequence of initial values V. The returned object is a function whose domain is the integers.

```
HilbertSeries(p,V)
```

```
HilbertSeries(Q,V)
```

The Hilbert series determined by a univariate polynomial p (the Hilbert polynomial), or a sequence Q of univariate polynomials, and a sequence of initial values V. The returned object is a rational function in one variable.

117.2.2 Interpreting the Hilbert Numerator

The Hilbert series of a variety $X \subset \mathbf{P}(w_0, \ldots, w_N)$ defined by equations of degrees $d_1, \ldots, d_r$ has the form

$$P(t) = \frac{1 - t^{d_1} - t^{d_2} - \cdots - t^{d_r} + t^e \pm \ldots \pm t^k}{(1 - t^{w_0}) \cdots (1 - t^{w_N})}.$$

We refer to the numerator of this expression as the *Hilbert numerator with respect to the weights* $w_0, \ldots, w_N$, or simply *Hilbert numerator* if the weights are clear.

The aim is to apply a converse statement: if we can manipulate the Hilbert series of a polarised variety into such a form, we guess that it is embedded in $\mathbf{P}(w_0, \ldots, w_N)$ by equations of degrees given by low degree terms with negative coefficients (where the absolute value of the coefficient determines the number of equations of that degree).

This can be taken further: the syzygies, and indeed a free resolution of the ideal of X, are also encoded weakly in the weights and the Hilbert numerator. For more details see [Rei00], Section 3, or [ABR02], Section 4.

```
HilbertSeriesMultipliedByMinimalDenominator(p,V)
```

```
HilbertSeriesMultipliedByMinimalDenominator(Q,V)
```

Let h be the Hilbert series determined by a univariate polynomial p, or a sequence Q of univariate polynomials, and a sequence of initial values V. This function produces a rational expression $g(t)/\Pi(1-t^d)$ for h, returning the numerator g as first value and a sequence containing the factors $1-t^d$ of the denominator as second value.

```
HilbertNumerator(g, D)
```

The product $g \times \Pi(1-t^d)$ taken over all integers $d \in D$, assuming the result is a polynomial. This function is typically used when g is a Hilbert series and D is a proposed collection of weights for a variety realising this series.

Example H117E1

We reproduce the example described in the introduction to this Section 117.2.

```
> T<t> := PolynomialRing(Rationals());
> p := 2*t - 3;
> V := [ 1, 2, 3, 4 ];
> h_fun := HilbertFunction(p,V);
> [ h_fun(n) : n in [0..7]];
[ 1, 2, 3, 4, 5, 7, 9, 11 ]
> h := HilbertSeries(p,V);
> h;
(t^5 + 1)/(t^2 - 2*t + 1)
```

The Hilbert series will be the Hilbert series of a polarised curve C (since the degree of the Hilbert polynomial—the order of growth of the coefficients of the Hilbert series—is one). It is already expressed as a rational function, so we apply a power series ring coercion to see the dimensions of the graded pieces—we could increase the precision if required.

```
> S<s> := PowerSeriesRing(Rationals(),8);
> S ! h;
1 + 2*s + 3*s^2 + 4*s^3 + 5*s^4 + 7*s^5 + 9*s^6 + 11*s^7 + O(s^8)
```

The rule of thumb for interpreting Hilbert series as varieties defined by homogeneous polynomials in wps requires one to write the Hilbert series in the form P/Q for polynomials P, Q, where Q is a product of terms $1-t^a$ (corresponding to the coordinates of weight a). The next line computes the Hilbert series together with the minimal such expression.

```
> HilbertSeriesMultipliedByMinimalDenominator(p,V);
t^5 + 1
[ -t + 1, -t + 1 ]
```

One could interpret this return value as a Noether normalisation of the graded ring of C. Instead, we try some values for a new weight a hoping to stumble on the weights of a wps in which C embeds. We start with $a = 4$, then try $a = 5$.

```
> HilbertNumerator(h,[1,1,4]);
-t^9 + t^5 - t^4 + 1
```

```
> HilbertNumerator(h,[1,1,5]);
-t^10 + 1
```

The first answer made little sense. Having included a generator in degree 4 it demanded an equation of the same degree. That is certainly possible, but assuming the equation involves only the weight 1 variables, it would factorise and the result would not be a variety. However, the final answer is just what we hope for: it suggests that C is realised by a degree 10 curve in $\mathbf{P}(1,1,5)$, which makes perfect sense and is what we expected.

Now consider the modified Hilbert series of the second example of Section 117.2 in which we changed the early coefficients from $V = [1,2,3,4]$ to $[1,1,2,4]$, but kept the Hilbert polynomial p.

```
> h1 := HilbertSeries(p,[1,1,2,4]);
> S ! h1;
1 + s + 2*s^2 + 4*s^3 + 5*s^4 + 7*s^5 + 9*s^6 + 11*s^7 + O(s^8)
> HilbertNumerator(h1,[1,1,5]);
-t^10 + t^9 - t^8 - t^7 + t^6 - t^4 + t^3 + t^2 - t + 1
```

This implies that there is an equation in degree 1. Such an equation immediately eliminates one of the two degree 1 coordinates which would be daft, so we eliminate this redundancy right at the beginning by trying instead

```
> HilbertNumerator(h1,[1,5]);
t^9 + t^7 + 2*t^6 + t^5 + t^4 + 2*t^3 + t^2 + 1
```

In short, this suggests a new variable in degree 2 and two new variables in degree 3: this could be done methodically by considering one generator at a time.

```
> HilbertNumerator(h1,[1,2,3,3,5]);
-t^17 + t^12 + 2*t^11 - 2*t^6 - t^5 + 1
```

Now examining this Hilbert numerator, we see that the first equation is in degree 5. But there is a variable of degree 5. Although that is not a problem, we could try to remove the degree 5 variable, as we would, in practice, if we knew that it appeared linearly in the equation.

```
> HilbertNumerator(h1,[1,2,3,3]);
t^12 - 2*t^6 + 1
```

Finally, this suggests $B_{6,6} \subset \mathbf{P}(1,2,3,3)$, and again it is an easy exercise in wps to confirm that such a curve really does exist with the predicted properties.

```
FindFirstGenerators(g)
```

This intrinsic function returns a sequence containing the results of a first attempt to deduce plausible weights of generators for the variety with Hilbert series g. The method used proceeds by advancing through the coefficients of g, in order of increasing degree, adding generators of that degree for each positive coefficient. The algorithm stops when it first finds a negative coefficient.

Example H117E2

We first make a Hilbert series which has initial terms $1 + 2t + 3t^2 + 4t^3$.

```
> T<t> := PolynomialRing(Rationals());
> p := 2*t - 3;
> V := [ 1, 2, 3, 4 ];
> h := HilbertSeries(p,V);
> h;
(t^5 + 1)/(t^2 - 2*t + 1)
> S<s> := PowerSeriesRing(Rationals(),4);
> S ! h;
1 + 2*s + 3*s^2 + 4*s^3 + O(s^4)
```

This h is, in fact, the Hilbert series of a graded ring generated by three elements in degrees 1,1,5.

```
> FindFirstGenerators(h);
[ 1, 1, 5 ]
```

There is no guarantee that this intrinsic will always find suitable weights, but it does return a subset of the weights that must occur for any variety that realises the given Hilbert series.

```
ApparentCodimension(f)
ApparentEquationDegrees(f)
ApparentSyzygyDegrees(f)
```

If $f = f(t)$ is a polynomial of the form

$$f = 1 - \sum_{i=1}^{N_0} a_{0,i}t^i + \sum_{k=N_0+1}^{N_1} a_{1,i}t^i - \ldots + (-1)^{k-1} \sum_{i=N_{k-2}+1}^{N_{k-1}} a_{k-1,i}t^i + (-1)^k t^{N_k}$$

then the apparent codimension is k, the apparent equation degrees are those i for which $a_{0,i}$ is nonzero (with $a_{0,i}$ equations having that degree i) and the apparent syzygy degrees are those i for which $a_{1,i}$ is nonzero.

117.3 Baskets of Singularities

We describe how to create the various point and curve singularities, together with collections or *baskets* of these singularities that are an ingredient of RR in higher dimensions.

Recall the basic principle of singularity contributions in RR: when we nominate a basket of singularities in the RR formula we do not necessarily expect the corresponding variety to have exactly those singularities associated with it (although that is very commonly the case), but rather to possess singularities which make exactly the same contribution to the RR formula as the singularities of the basket. Thus, we only allow the creation of a restricted class of singularities that (in good cases) allow us to realise all possible contributions.

117.3.1 Point Singularities

The point singularities that are allowed are always finite cyclic quotient singularities. In the surface case (over the complex numbers **C**) they are always analytically isomorphic to a neighbourhood of the origin in the quotient $\mathbf{C}^2/G$, where G is the group of r-th roots of unity and the action is determined on eigencoordinates x, y of $\mathbf{C}^2$ by the action of a primitive root of unity $\lambda \in G$:

$$\lambda \cdot x = \lambda^a x, \quad \lambda \cdot y = \lambda^b y$$

for some $a, b \in \{0, \dots, r-1\}$. Such a singularity is denoted by the symbol $\frac{1}{r}(a, b)$.

Similarly, one defines 3-dimensional quotient singularities, where the symbol $\frac{1}{r}(a, b, c)$ denotes a cyclic quotient singularity. Of course, one can go into yet higher dimensions, with symbol $\frac{1}{r}(a_1, \dots, a_k)$, but although the functions of this chapter can create such singularities, they do not yet calculate RR for singularities of dimension higher than three. By definition, if $p = \frac{1}{r}(a_1, \dots, a_k)$, the *index of* p is the positive integer r and the *polarisation of* p is the sequence $[a_1, \dots, a_k]$.

The four main situations in which we use (quotient) point singularities are:

- Gorenstein surface singularities $\frac{1}{r}(a, r-a)$ with r coprime to a (for K3 surfaces)
- terminal 3-fold singularities $\frac{1}{r}(a, r-a, b)$ with r coprime to a, b (for Fano 3-folds)
- isolated canonical 3-fold singularities $\frac{1}{r}(a, b, c)$ with $a+b+c = 0 \bmod r$ and r coprime to each of a, b, c (for Calabi–Yau 3-folds)
- nonisolated canonical 3-fold singularities $\frac{1}{r}(a, b, c)$ with $a+b+c = 0 \bmod r$ and no three of r, a, b, c sharing a nontrivial common factor (for Calabi–Yau 3-folds).

In the final, nonisolated case, the points can be points at the intersection of two branches of the 1-dimensional singular loci, or other points on 1-dimensional singular loci that do not have the generic transverse behaviour. (This case is discussed further in Section 104.1.1 on curve singularities below.)

There is an additional key piece of data. The contribution of a point p to RR for $\chi(X, A)$ depends upon the eigenspace of the G-action in which A lies. In other words, the singularity is also polarised locally by A. This is called the *local polarisation* or the *eigenspace* of the singularity (the latter to distinguish it more clearly from the polarisation), and is another integer n in the range $\{0, \dots, r-1\}$. When we need to include the eigenspace of A in the singularity, we use the symbol $\frac{1}{r}(a_1, \dots, a_k)_n$.

Example H117E3

We make a point that can lie on a surface.

```
> p := Point(7,[3,4]);
> p;
1/7(3,4)
> IsGorensteinSurface(p);
true
```

Now we make a point of a 3-fold.

```
> q := Point(5,2,[1,2,3]);
> q;
1/5(1,2,3)_[2]
> IsTerminalThreefold(q);
true
> Eigenspace(q);
2
> p eq q;
false
```

We did not assign an eigenspace to the point p, so a default value was assigned.

```
> Eigenspace(p);
-1
```

117.3.1.1 Creation of Point Singularities

```
Point(r,n,Q)
```

```
Point(r,Q)
```

The point singularity with index a positive integer r, polarisation a sequence of positive integers Q and local polarisation an integer n. The group action should have no quasi-reflections, which means that no integer $k > 1$ is allowed to divide r and all but one of the elements of Q. The local polarisation n modulo r should be a unit modulo r. If it is not given as an argument, the default value is $n = -1$.

117.3.1.2 Accessing the Key Data and Testing Equality

```
Dimension(p)
```

Dimension of the variety on which the singularity p lies.

```
Index(p)
```

Index of the singularity p.

```
Polarisation(p)
```

Polarisation sequence of the singularity p.

```
Eigenspace(p)
```

Eigenspace of the polarising divisor of the point singularity p.

```
p eq q
```

Return `true` if and only if the two point singularities p, q have the following attributes equal: dimension, index, polarisation, eigenspace.

117.3.1.3 Identifying Special Types of Point Singularity

```
IsIsolated(p)
```

Return `true` if and only if the point singularity p is an isolated singularity; that is, if and only if every element of its polarisation is coprime to its index.

```
IsGorensteinSurface(p)
```

Return `true` if and only if the point singularity p is a Gorenstein surface point; that is, if and only if p is of type $\frac{1}{r}(a, r-a)$ for r, a coprime.

```
IsTerminalThreefold(p)
```

Return `true` if and only if the point singularity p is a terminal 3-fold point; that is, if and only if it is isolated, dimension 3 and the polarisation is of the form $a, b, r-b$, up to a permutation.

```
TerminalIndex(p)
```

The integer a when p is a singular point with polarisation sequence of the form $a, b, r-b$, up to a permutation, assuming that p is a terminal 3-fold point singularity.

```
TerminalPolarisation(p)
```

Polarisation sequence of the singularity p in the order $a, b, r-b$, where r is the index of p, and $b \leq r/2$. There will be an error if p is not a terminal 3-fold point singularity.

```
IsCanonical(p)
```

Return `true` if and only if the point singularity p is canonical; that is, if and only if the sum of the polarisation is trivial modulo the index.

117.3.2 Curve Singularities

We follow Buckley's analysis of curve singularities in RR [Buc03]. The first thing to note is that curve singularities in Magma are always 3-dimensional, that is, they are one-dimensional singular loci $C \subset X$ of polarised 3-folds X, A. The *degree* of C is the intersection number AC. At a general point of a curve singularity C, a transverse section is a Gorenstein surface quotient singularity, that is, a point of type $\frac{1}{r}(a, r-a)$ for coprime integers r, a. This is called the *transverse type* of C and r is called the *transverse index* of C.

There are two further key attributes of a curve singularity C: two integers N, t that carry information about the normal bundle of C and about possible special points on C. We do not discuss the meaning of N here except to say that it encodes the splitting type of the normal bundle (see [Buc03] and [SB] for further information), but in any case note that there are intrinsics described below that can identify appropriate values for N given some other invariants. This pair of invariants occur in RR only in the combination N/t, which itself appears only linearly. So given enough coefficients of the Hilbert series as input, this value can be recovered. A curve, therefore, does have an attribute 'magic' that records this magic number N/t, and this can be set (and used in RR) even if N, t are not assigned.

We explain the attribute t. The transverse type of C gives a transverse section at any general point along C. But there may be special points, so-called *dissident points*, that do not have such a section. They certainly occur at an intersection point of two curve singularities, but may also be other points in C. They are quotient singularities $\{p_i\}$ whose indices are of the form rt_i for positive integers t_i, where r is the transverse index of C. The invariant t, called the *index* of C, is the GCD of the set of all such t_i.

Example H117E4

We create a curve singularity with given transverse type.

```
> p := Point(3,[1,2]);
> C := Curve(1/3,p,4,3);
> C;
Curve of degree 1/3, N = 4, t = 3 with transverse type 1/3(1,2)
```

We check some of the characteristics of C.

```
> TransverseIndex(C);
3
> IsCanonical(C);
true
> MagicNumber(C);
4/3
```

117.3.2.1 Creation of Curve Singularities

```
Curve(d,p,m)
Curve(d,p,N)
Curve(d,p,N,t)
```

The 3-fold curve singularity of degree d, transverse type p (a point surface singularity) and characteristic numbers N, t (which is 1 if not set) or magic number $m = N/t$.

117.3.2.2 Accessing the Key Data and Testing Equality

```
Degree(C)
```

The degree of the curve singularity C.

```
TransverseType(C)
```

The transverse type of the curve singularity C, that is, a point surface singularity that is the general transverse section of C.

```
TransverseIndex(C)
```

The transverse index of the curve singularity C, that is, the index of the point surface singularity that is a general transverse section of C.

```
NormalNumber(C)
```

The invariant N of the curve singularity C.

```
Index(C)
```

The invariant t (sometimes called τ) of the curve singularity C that is determined by the indices of dissident points on C.

```
MagicNumber(C)
```

The number N/t of the curve singularity C, where N is the normal number of C and t is the index of C.

```
Dimension(C)
```

The dimension of the curve singularity C (currently always 3).

```
IsCanonical(C)
```

Return `true` if and only if the transverse type of the curve singularity C is a canonical (or Du Val) surface singularity.

```
C eq D
```

Return `true` if and only if the two curve singularities C, D have the following attributes equal: dimension, degree, transverse type, magic number (or, equivalently, the pair of invariants N, t).

117.3.3 Baskets of Singularities

A *basket of singularities*, or simply *basket*, is a collection of point and curve singularities. One constructs baskets in the hope of finding a variety that has exactly those singularities lying on it as its only singularities, and very often this is indeed the case. It is a marginal issue here, but worth noting that in fact baskets are collections of ideal singularities of a variety X that make exactly the same contributions to RR as the actual singularities of X. So as a matter of principle, one is not primarily seeking varieties with exactly the basket singularities, even though this is what happens in practice.

117.3.3.1 Creation and Modification of Baskets

```
Basket(Q)
```

```
Basket(Q1,Q2)
```

The basket of singularities where Q, Q_1, Q_2 are sequences of point or curve singularities.

```
EmptyBasket()
```

The basket of singularities containing no singularities.

```
MakeBasket(Q)
```

The basket of singularities containing point singularities that are encoded in the sequence Q. This may occur in different ways. If Q is a sequence of sequences of the form $[r, a]$ (that is, each having length 2) with r, a coprime, then the result will be a basket of points of the form $\frac{1}{r}(a, r - a)$. If Q is a sequence of sequences of some common length $N > 2$, then the result will be a basket of points of the form $\frac{1}{Q[1]}(Q[2], \dots, Q[N])$. Note that a local polarisation n cannot be included in this constructor: the default value $n = -1$ is always assumed.

```
Points(B)
```

The sequence of point singularities of the basket B.

```
Curves(B)
```

The sequence of curve singularities of the basket B.

117.3.3.2 Tests for Baskets

```
IsIsolated(B)
```

Return `true` if and only if all singularities of the basket B are isolated (so, in particular, there are no curve singularities in B).

```
IsGorensteinSurface(B)
```

Return `true` if and only if all singularities of the basket B are Du Val singularities, that is, are of the form $\frac{1}{r}(a, r - a)$.

```
IsTerminalThreefold(B)
```

Return `true` if and only if all singularities of the basket B are terminal 3-fold singularities.

```
IsCanonical(B)
```

Return `true` if and only if all singularities of the basket B are canonical singularities.

117.3.4 Curves and Dissident Points

Curves of singularities may contain special points not of the typical transverse type; these are the dissident points. The following intrinsics generate sequences of possible dissident points for a particular curve singularity. The point is that some of the invariants of a curve singularity force a curve to have dissident points that together give a certain RR contribution. It is not always easy to see what these points should be, so constructing by hand baskets that include a particular curve singularity can be difficult. These intrinsics all give suggestions for possible sets of dissident points.

```
CanonicalDissidentPoints(C)
```

A sequence of sequences of points, each of which minimally accounts for the index of the curve singularity C (although further curves may be needed in order for it to make sense).

```
SimpleCanonicalDissidentPoints(C)
```

A sequence of sequences of points, each of which minimally accounts for the index of the curve singularity C and which does not allow further curves to meet C.

```
PossibleCanonicalDissidentPoints(C)
```

A sequence of points, each of may appear on the curve singularity C as a dissident point.

```
PossibleSimpleCanonicalDissidentPoints(C)
```

A sequence of points, each of may appear on the curve singularity C as a dissident point but which is not at the intersection of C with another curve.

117.4 Generic Polarised Varieties

Recall from Section 117.1.1 that, despite some ambiguity, we regard the following as being equivalent: polarised varieties X, A; schemes in wps $X \subset \mathbf{P}^N(w_0, \ldots, w_N)$ where A is a degree 1 hyperplane section; and data about the Hilbert series of the graded ring $R(X, A)$. Thus we constantly refer to a *polarised variety* X, and we expect to be able to retrieve its Hilbert series, its dimension, the codimension of its embedding and other such data.

With one exception, the intrinsics described in this section can be applied to all polarised varieties. The exception is the following little-used intrinsic that creates a polarised variety that is not of a specific type.

```
PolarisedVariety(d,W,n)
```

The polarised variety of dimension d, with weights given by the sequence W of positive integers and with Hilbert numerator the univariate polynomial n.

117.4.1 Accessing the Data

```
Weights(X)
```

The weights of the polarised variety X.

```
Degree(X)
```

The degree of the polarised variety X.

```
Basket(X)
```

The basket of singularities of the polarised variety X.

```
RawBasket(X)
```

The basket of singularities of the polarised variety X in sequence format, that is, the basket is a sequence of sequences, in which a singularity $\frac{1}{r}(a, b, c)$ is represented as a sequence $[r, a, b, c]$ of integers. (The Gorenstein surface singularity $\frac{1}{r}(a, r - a)$ admits further abbreviation to $[r, a]$.) Notice that the local polarisation n is not included in this raw basket data; its default value $n = -1$ is assumed.

```
Dimension(X)
```

The dimension of the polarised variety X.

```
Codimension(X)
```

The codimension of the polarised variety X.

```
HilbertNumerator(X)
```

```
Numerator(X)
```

The numerator $f(t)$ of the Hilbert series $P(t)$ of the polarised variety X when expressed as a rational function $P = f(t)/\& * [1 - t^w : w \in W]$ where the product in the denominator is taken over W, the sequence of weights of X.

```
NoetherWeights(X)
```

The weights corresponding to a Noether normalisation of the polarised variety X. In other words, these are the weights of polynomials in the graded ring of X that generate a polynomial subring of maximal dimension.

```
NoetherNumerator(X)
```

The numerator $n(t)$ of the Hilbert series $P(t)$ of the polarised variety X when expressed as a rational function $P = n(t)/\& * [1 - t^w : w \in N]$, where the product in the denominator is taken over N, the sequence of Noether weights of X.

```
NoetherNormalisation(X)
```

Given a polarised variety X return a pair, the first term of which is the sequence of Noether weights, the second the corresponding numerator.

```
HilbertSeries(X)
```

The Hilbert series of the polarised variety X expressed as a rational function.

```
InitialCoefficients(X)
```

The coefficients of the Hilbert series of the polarised variety X expressed as a power series. The number of coefficients returned is equal to the precision of the power series ring in which the Hilbert series was expanded.

```
ApparentCodimension(X)
ApparentEquationDegrees(X)
ApparentSyzygyDegrees(X)
BettiNumbers(X)
```

If $n(t)$ is the Hilbert numerator of X and is of the form

$$n = 1 - \sum_{i=1}^{N_0} a_{0,i}t^i + \sum_{k=N_0+1}^{N_1} a_{1,i}t^i - \ldots + (-1)^{k-1} \sum_{i=N_{k-2}+1}^{N_{k-1}} a_{k-1,i}t^i + (-1)^k t^{N_k}$$

then the apparent codimension of X is k, the apparent equation degrees are given by those i for which $a_{0,i}$ is nonzero (with $a_{0,i}$ equations of that degree i) and the apparent syzygy degrees are those integers i for which $a_{1,i}$ is nonzero. The Betti numbers are a sequence with first element the sum of all $a_{0,i}$, second element the sum of all $a_{1,i}$, and so on until $a_{k-1,i}$.

117.4.2 Generic Creation, Checking, Changing

Procedural versions of intrinsic functions modify polarised varieties at the generic level because they preserve any subtypes; functional versions exist for special types of polarised variety but not in general.

```
X eq Y
```

Return `true` if and only if the polarised varieties X and Y have the same dimension, weights, basket and Hilbert numerator. In particular, these conditions imply that X and Y have the same Hilbert series.

```
CheckCodimension(X)
```

Return `true` if and only if the codimension of X is equal to the apparent codimension of X determined by its Hilbert numerator.

```
FirstWeights(X)
```

These are weights assigned to the polarised variety X during its construction that carry some relevance; if no such weights were assigned, the usual weights of X will be returned.

```
IncludeWeight(~X,w)
```

Include the positive integer w among the weights of X, adjusting all other data associated to the embedding of X as required.

```
RemoveWeight(~X,w)
```

Remove the positive integer w from the weights of X, assuming it appears there and can be removed without destroying the property of the Hilbert numerator being a polynomial. All other data associated to the embedding of X is modified as required.

```
MinimiseWeights(~X)
```

Remove any weights from X whose presence is not required to keep the Hilbert numerator of X a polynomial.

117.5 Subcanonical Curves

A *subcanonical curve* is a polarised variety C, D where C is a nonsingular curve of genus $g \geq 2$ and D is a divisor on C such that $K_C = kD$ for some positive integer k.

117.5.1 Creation of Subcanonical Curves

```
SubcanonicalCurve(g,d,Q)
```

The subcanonical curve C, D of genus g, degree d and initial Hilbert series coefficients Q.

```
IsSubcanonicalCurve(g,d,Q)
```

Return `true` if and only if the data g, d, Q passes some basic checks that there is a subcanonical curve C, D of genus g, degree d and initial Hilbert series coefficients Q. In that case, the second return value is such a curve.

```
HilbertPolynomialOfCurve(g,m)
```

The Hilbert polynomial $mt + 1 - g$ of a divisor of degree m on a curve of genus g.

```
IsEffective(C)
```

Return `true` if and only if the polarising divisor of the subcanonical curve C is effective; that is, if and only if the Hilbert series has the form $1 + p_1 t + \cdots$ with $p_1 > 0$.

117.5.2 Catalogue of Subcanonical Curves

This section describes intrinsics that allow the user to generate many examples of Hilbert series of subcanonical curves and attempt to interpret them as curves embedded in wps.

```
EffectiveSubcanonicalCurves(g)
```

```
EffectiveSubcanonicalCurves(g,d)
```

A sequence containing data for effective subcanonical curves of genus $g \geq 3$ (polarised by a divisor of degree d if the second argument is given).

```
IneffectiveSubcanonicalCurves(g)
```

```
IneffectiveSubcanonicalCurves(g,d)
```

A sequence containing data for ineffective subcanonical curves of genus $g \geq 3$ (polarised by a divisor of degree d if the second argument is given).

117.6 K3 Surfaces

This section describes intrinsics that construct K3 surfaces. It also describes a few intrinsics that can be used to study them, but see Section 117.4 for the general intrinsics that apply to all polarised varieties.

The calculations are based on Altinok's Riemann–Roch formula [Alt98] for polarised K3 surfaces with Du Val singularities.

117.6.1 Creating and Comparing K3 Surfaces

The basic RR data from which a K3 surface can be created comprises an integer, the *genus*, $g \geq -1$ and a basket of (Gorenstein surface) point singularities B. The basket B can be created explicitly as a basket using the functions of Section 104.1.1, but a convenient shortcut is provided whereby the basket argument may be given 'in raw basket format', that is, as a sequence B of length two sequences, each of the form $[r, a]$, denoting the singularity $\frac{1}{r}(a, r-a)$.

```
K3Surface(g,B)
```

A K3 surface with genus g and basket of singularities B (which may be a basket type or in raw basket format $[[r, a], ...]$).

```
K3Copy(X)
```

A new K3 surface that carries exactly the same data as the K3 surface X.

117.6.2 Accessing the Key Data

```
Genus(X)
```

The genus of the K3 surface X; that is, $p_1 - 1$ where p_1 is the coefficient of t in the Hilbert series of X.

```
TwoGenus(X)
```

The 2-genus of the K3 surface X; that is, $p_2 - 1$ where p_2 is the coefficient of t^2 in the Hilbert series of X.

```
SingularRank(X)
```

The sum $\sum(r-1)$ taken over the singularities $\frac{1}{r}(a, r-a)$ given in the basket of singularities of the K3 surface X.

```
AFRNumber(X)
```

The number assigned to the K3 surface X in the low codimension lists of Altinok–Fletcher–Reid.

117.6.3 Modifying K3 Surfaces

Sometimes it is desirable to add or remove weights from a given K3 surface. There are two intrinsics that allow this to be done (and check that a weight really can be removed). These intrinsics are used systematically in the construction of the K3 database.

```
IncludeWeight(X,w)
```

Return a new K3 surface that is the same as X but with the positive integer w included among the weights and all other data associated to the embedding adjusted as required.

```
RemoveWeight(X,w)
```

Return a new K3 surface that is the same as X but with the positive integer w removed from the weights, assuming it appears there and can be removed without destroying the property of the Hilbert numerator being a polynomial. All other data associated to the embedding is adjusted as required.

117.7 The K3 Database

The K3 database in Magma is a collection of 24,099 K3 surfaces. Recall from Section 117.6 the meaning of K3 surface in this context, and from Section 117.1.1 the relationship between the Hilbert series, the weights and the (Hilbert) numerator.

We describe the set of K3 surfaces selected for inclusion in the database. For $g = -1, 0, 1, 2$, all K3 surfaces of genus g are included, there being 4281, 6479, 6627 and 6628 surfaces, respectively. For higher genus, the data associated to the 6628 K3 surfaces of genus 2 propagates in a predictable way, so only those K3 surfaces with codimension at most 7 and genus in the range 3 to 9 have been included.

Data is held in blocks of surfaces indexed by the first five coefficients of their Hilbert series (excluding the constant term). Note that the t-coefficient of the Hilbert series is one more that the genus, and this defect holds for all genera. To determine the number of surfaces of genus 1, the intrinsics described below may be used. Note the genus argument is a sequence beginning with the integer 2: the sequence is arranged so that the user can ask a more precise question by including other leading genera (up to the first five), and the value 2 is to account for the genus–Hilbert coefficient defect.

```
> D := K3Database();
> NumberOfK3Surfaces(D,[2]);
6627
```

The database is fairly large, so naive searches take time. Specialised tools, described below, support much more efficient searches and should be used wherever possible. We demonstrate this point with timings for a typical search. The first searches the entire database for all K3 surfaces of genus 3 and takes over 2 minutes. It is much more efficient to use a function that looks up curves according to their genus, since this is the primary indexing property used by the database. The second search does this, and takes only a fraction of the time.

```
> time [ X : X in D | Genus(X) eq 3 ];
Time: 139.510
> time [K3SurfaceK(D,[4],i) :i in [1..NumberOfK3Surfaces(D,[4])]];
Time: 0.500
```

117.7.1 Searching the K3 Database

In this section a simple example is presented of extracting a K3 surface with particular properties from the K3 database. Section 104.1.1 provides much greater details and more examples: note, in particular, that only a few hundred of the surfaces that occur in small codimension have been confirmed to exist (even though the vast majority are believed to exist).

Example H117E5

We begin by defining D to be the K3 database.

```
> D := K3Database();
> D;
The database of K3 surfaces
```

It contains data associated to 24099 (families of) K3 surfaces.

```
> #D;
24099
```

There are several ways to access the K3 surfaces in the database. In the first place, the database is organised into blocks of K3 surfaces that have a common genus. These blocks are then subdivided into K3 surfaces that have a common 2-genus. The blocks having a common 2-genus are further subdivided right down to 5-genus, that is, the coefficient of t^5 in the Hilbert series. These subdivisions are the natural indexing units of the database. One gets the third surface with genus 0 by

```
> X := K3Surface(D,0,3);
> X;
K3 surface no.3, genus 0, in codimension 1 with data
  Weights: [ 1, 6, 8, 9 ]
  Basket: 1/2(1,1), 1/3(1,2), 1/9(1,8)
  Degree: 1/18          Singular rank: 11
  Numerator: -t^24 + 1
  Projection to codim 1 K3 no.2 -- type I from 1/9(1,8)
  Unproj'n from codim 2 K3 no.4 -- type I from 1/10(1,9)
  Unproj'n from codim 2 K3 no.15 -- type IV from 1/5(2,3)
  Unproj'n from codim 3 K3 no.28 -- type II_1 from 1/4(1,3)
  Unproj'n from codim 4 K3 no.84 -- type II_2 from 1/3(1,2)
  Unproj'n from codim 6 K3 no.280 -- type II_5 from 1/2(1,1)
```

This printout displays a lot of information about this surface and its relationship to other surfaces. The minimal printing option may be use to obtain a concise description of this surface alone.

```
> X:Minimal;
K3 surface (g=0, no.3) in P^3(1,6,8,9)
  Basket: 1/2(1,1), 1/3(1,2), 1/9(1,8)
  Numerator: -t^24 + 1
```

When using several genera to access a surface, the genus arguments must be collected together in a sequence. For example, there are 282 K3 surfaces whose first three genera are $p_1 = 0$, $p_2 = 1$, $p_3 = 3$; that is, have weights that are of the form $[2, 3, 3, \ldots]$.

```
> NumberOfK3Surfaces(D,[0,1,3]);
282
```

We get the first of these as follows. The arguments inside the sequence brackets are coefficients of the Hilbert polynomial, while the corresponding genus is one less than the coefficient. N.B. Note the offset by -1 between these arguments and the genera.

```
> K3Surface(D,[0,1,3],1);
K3 surface no.1130, genus -1, in codimension 4 with data
  Weights: [ 2, 3, 3, 3, 4, 4, 5 ]
  Basket: 2 x 1/2(1,1), 5 x 1/3(1,2)
  Degree: 1/3          Singular rank: 12
  Numerator: t^24 - ... + t^10 - t^9 - 2*t^8 - t^7 - t^6 + 1
  Projection to codim 1 K3 no.820 -- type II_2 from 1/3(1,2)
  Unproj'n from codim 5 K3 no.1131 -- type I from 1/5(2,3)
  Unproj'n from codim 6 K3 no.1145 -- type II_1 from 1/4(1,3)
  Unproj'n from codim 7 K3 no.1412 -- type II_2 from 1/3(1,2)
  Unproj'n from codim 8 K3 no.2176 -- type IV from 1/2(1,1)
```

The genus and number of a K3 surface identifies it uniquely in the database, so the same function may be used to see surface number 1131 which has projection to X.

```
> K3Surface(D,-1,1131) : Minimal;
K3 surface (g=-1, no.1131) in P^7(2,3,3,3,4,4,5,5)
  Basket: 1/2(1,1), 4 x 1/3(1,2), 1/5(2,3)
  Numerator: -t^29 + ... + 6*t^11 - 3*t^9 - 4*t^8 - t^7 - t^6 + 1
```

The projection is from the $\frac{1}{5}(2,3)$ singularity, resulting in the extra $\frac{1}{2}(1,1)$ and $\frac{1}{3}(1,2)$ points. There are also searches that do not use the primary indexing directly. For example, the following variation of `K3Surface` searches for a K3 surface with weights $2,2,3,5,7,9,11$.

```
> K3Surface(D,[2,2,3,5,7,9,11]) : Minimal;
K3 surface (g=-1, no.1615) in P^6(2,2,3,5,7,9,11)
  Basket: 3 x 1/2(1,1), 1/11(2,9)
  Numerator: t^39 - ... + t^16 - t^13 - t^11 - t^9 + 1
```

`K3Database()`

The database of K3 surfaces.

`Number(D,X)`

The integer n such that the K3 surface `Y := K3Surface(D,Genus(X)+1,n)` in the database D has the same Hilbert series as the K3 surface X. The second return value is the K3 surface Y. If there is no such K3 surface, the returned index value is zero.

`Index(D,X)`

The integer i such that the K3 surface `Y := K3Surface(D,i)` in the database D has the same Hilbert series as the K3 surface X. The second return value is the K3 surface Y. If there is no such K3 surface, the returned index value is zero.

Example H117E6

The ‘Number’ of a K3 surface in the database and its ‘Index’ may differ: the K3 surfaces of any fixed genus are numbered separately, while the index runs over the whole database.
To illustrate this, consider the following K3 surface.

```
> X := K3Surface(1,[[2,1],[3,1],[4,1],[7,1],[8,1]]);
> X;
K3 surface in codimension 11 with data
  Weights: [ 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 6, 7, 8 ]
  Basket: 1/2(1,1), 1/3(1,2), 1/4(1,3), 1/7(1,6), 1/8(1,7)
  Degree: 613/168             Singular rank: 19
  Numerator: -t^47 + ... + 38*t^7 - 2*t^6 - 13*t^5 - 7*t^4 + 1
```

This surface has been calculated in isolation. The weights that have been assigned to it are just enough to make sense of the initial terms of the Hilbert series, and to make the singularities. By construction, the K3 database may have added weights to make simple projections work. So we search for X in the database using either ‘Index’ or ‘Number’.

```
> D := K3Database();
> n,Y := Number(D,X);
> i,Y1 := Index(D,X);
> n,i;
1474 12234
```

This result means that X has the same Hilbert series as the 1474-th K3 surface in D of genus 1, which is the same as the 12234-th K3 surface in D.

```
> Y eq Y1;
true
> Y1 eq K3Surface(D,i);
true
```

One can see that Y is in much higher codimension than X, so that extra weights have been assigned to Y.

```
> Codimension(Y);
17
```

117.7.2 Working with the K3 Database

```
K3Surface(D,i)
```

The ith K3 surface in the K3 database D.

```
K3Surface(D,Q,i)
```

The ith K3 surface in the K3 database D among those with index suite $Q = [g1 + 1, g2 + 1, ...]$.

`K3Surface(D,g,i)`

The ith K3 surface in the K3 database D among those with genus $g \geq -1$.

`K3Surface(D,g1,g2,i)`

The ith K3 surface in the K3 database D among those with genus $g1 \geq -1$ and 2-genus $g2 \geq -1$.

`K3Surface(D,W)`

The K3 surface in the K3 database D having the weights specified in the sequence W.

`K3Surface(D,g,B)`

The K3 surface in the K3 database D with genus $g \geq -1$ and basket of singularities B (as a basket type or in raw sequence format).

117.8 Fano 3-folds

This section describes intrinsics that construct Fano 3-folds. It also describes a few intrinsics that can be used to study them, but see Section 117.4 for the general intrinsics that apply to all polarised varieties.

The calculations are based on Suzuki's Riemann–Roch formula [Suz] for polarised Fano 3-folds with terminal singularities.

Example H117E7

We make two Fano 3-folds having the same basket but different genus.

```
> X := Fano(2,MakeBasket([[3,1,2,2]]),2);
> X;
Fano 3-fold X,A of Fano index 2, Fano genus 2, in codimension 1 with data
  Weights: [ 1, 1, 2, 3, 5 ]
  Basket: 1/3(1,2,2)
  Degrees: A^3 = 1/3,   (1/12)Ac_2(X) = 8/9
  Numerator: -t^10 + 1
> FanoGenus(X);
2
> FanoBaseGenus(X);
2
> Fano(2,MakeBasket([[3,1,2,2]]),3);
Fano 3-fold X,A of Fano index 2, Fano genus 3, in codimension 2 with data
  Weights: [ 1, 1, 1, 2, 2, 3 ]
  Basket: 1/3(1,2,2)
  Degrees: A^3 = 4/3,   (1/12)Ac_2(X) = 8/9
  Numerator: t^8 - 2*t^4 + 1
```

In this example, the smallest possible genus—the Fano base genus—is 2, and an error will be reported if a smaller value is requested.

Note that the singularities used must be polarised by fA, where f is the Fano index: in practice, this means their index r must be coprime to f and their weights must be of the form $f, a, r - a$.

117.8.1 Creation: $f = 1, 2$ or ≥ 3

```
Fano(f,B,g)
```

A Fano 3-fold with Fano index $f \geq 1$, Fano genus $g \geq 0$ and basket of singularities B. The singularities must be terminal singularities. The basket can also be presented in raw sequence format: in this case, B is a sequence containing terms such as $[r, a, b, c]$ which denotes the singularity $\frac{1}{r}(a, b, c)$.

```
Fano(f,B)
```

A Fano 3-fold with Fano index $f \geq 3$ and basket of singularities B. The singularities must be terminal singularities. The basket can also be presented in raw sequence format: in this case, B is a sequence containing terms such as $[r, a, b, c]$ which denotes the singularity $\frac{1}{r}(a, b, c)$.

```
FanoIndex(X)
```

The Fano index f of the Fano 3-fold X.

```
FanoGenus(X)
```

The Fano genus of the Fano 3-fold X, an integer ≥ 0 equal to the dimension of the space of sections of the polarising divisor. (The term *genus* often refers to two less that this number).

```
FanoBaseGenus(X)
```

The smallest possible value for the Fano genus of the Fano 3-fold X.

```
BogomolovNumber(X)
```

The intersection number $A(c_1(X)^2 - 3c_2(X))$ for the polarised Fano 3-fold X, A.

```
IsBogomolovUnstable(X)
```

Return `true` if and only if the Bogomolov number $A(c_1(X)^2 - 3c_2(X))$ for the polarised Fano 3-fold X, A is strictly positive.

117.8.2 A Preliminary Fano Database

```
FanoDatabase()
```

The database of Fano 3-folds.

```
Fano(D,i)
```

The ith Fano 3-fold in the Fano database D.

```
Fano(D,f,i)
```

The ith Fano 3-fold in the Fano database D that has Fano index f.

```
Fano(D,f,Q,i)
```

The ith Fano 3-fold in the Fano database D that has Fano index f and initial plurigenera as specified by the sequence Q (up to the first four plurigenera).

117.9 Calabi–Yau 3-folds

This section describes intrinsics that construct Calabi–Yau 3-folds. It also describes a few intrinsics that can be used to study them, but see Section 117.4 for the general intrinsics that apply to all polarised varieties.

The calculations are based on Buckley's Riemann–Roch formula [Buc03] for polarised Calabi–Yau 3-folds with canonical singularities.

```
CalabiYau(p1,p2,B)
```

The Calabi–Yau 3-fold X, A with $h^0(X, A) = p_1$, $h^0(X, 2A) = p_2$ and basket of singularities B.

```
FindN(X)
```

`MaximumN` RngIntElt *Default :* 100

The first nonnegative value, for the Calabi–Yau 3-fold X, of N_{C_i} (where C_i is the ith curve of the basket of X) together with the distance between successive values of N_{C_i}. (The pair $0, 0$ is returned if no solutions below the parameter `MaximumN` are found).

```
FindN(p1,p2,B)
```

`MaximumN` RngIntElt *Default :* 100

The first nonnegative value of N_{C_i} (where C_i is the ith curve of the basket B of 3-fold points and curves and p_1, p_2 are the first two genera) together with the distance between successive values of N_{C_i}. (The pair $0, 0$ is returned if no solutions below the parameter `MaximumN` are found).

117.10 Building Databases

This section contains a sketch of how databases of examples can be built. This is an aside from the rest of the chapter, and is only of interest if one either wants to understand the construction of the K3 database for its own sake, or wishes to create other similar lists that take too long to be regenerated on demand. Note, however, that the database facility described here is not suitable for very large databases: roughly speaking, it is intended for those containing a few tens of thousands of elements, not millions, and certainly not in the Kreuzer–Skarke [KS00] range.

117.10.1 The K3 Database

This section describes briefly the method of construction used for the K3 database. The K3 database is fully installed in Magma—see Section 117.7 for instructions—and there is no need to use the functions described here to rebuild it unless you intend to modify the code to incorporate new information in the database.

The construction is in two steps. First we create all the K3 surfaces we require and write them as abbreviated records to a series of files. Then we load these files one at a time, writing their contents in further abbreviated form to a binary data file, `K3S.dat`. The writing functions keep track of key indexing information and write this to an index file, `K3S.ind`. Functions to read from these files are already installed, so the process is completed by copying these two files to the standard data directory in the Magma libraries.

We now go through the process in detail.

117.10.1.1 Creating Many K3 Surfaces

```
CreateK3Data(g)
```

```
CreateK3Data(g,r)
```

```
CreateK3Data(g,B)
```

Create a sequence containing all K3 surfaces of genus $g \geq -1$; restrict to those with singular rank at most r, if given as an argument, or having baskets in the sequence B of baskets, if given as an argument.

The return sequence is ordered according to the natural numerical order on the coefficients of Hilbert series. Analysis of projections of Types I and II is made to modify weights of K3 surfaces, and any inconsistencies between different predictions of weights coming from different projections are reported.

117.10.1.2 K3 Surfaces as Records

```
K3SurfaceToRecord(X)
```

A record in K3 record format that contains a subset of the data associated to the K3 surface X from which all attributes of X can be computed.

```
K3Surface(x)
```

The K3 surface with the same data as that of the record x in K3 record format.

117.10.1.3 Writing K3 Surfaces to a File

```
WriteK3Data(Q,F)
```

Write the K3 surfaces in the sequence Q to the file with name given by the string F. The resulting file F is a text file containing Magma code which, when loaded into a Magma session, generates a sequence called `data` of records in K3 record format that corresponds to Q.

117.10.1.4 Writing the Data and Index Files

With the data for the K3 database saved as a series of files "k3data-1", "k3data0" and so on, the command `load "writek3db.m";` (or `load "PATH/writek3db.m";`) loads them in turn, writes the two binary files and then deletes the data from the Magma session. Magma must be running in the directory containing all the data files "k3data…".

117.10.1.5 Reading the Raw Data

In normal Magma use, even though the data in the K3 database is in coded form (as a tuple, in fact), when returned to the user it is expressed as a K3 surface. Of course, this final translation step takes a little time, insignificant in most use, but very significant when searching through the whole database. So there are functions to access the raw data, and then to translate it into a K3 surface. The search intrinsics installed in the Magma packages use these functions, as should any new searches that need only modest increase in speed.

```
K3SurfaceRaw(D,i)
K3SurfaceRaw(D,Q,i)
```

The i-th element of the K3 database D expressed as a tuple of data (or the i-th element whose Hilbert series has coefficients of t^i given by the integers in the sequence Q, which may have length at most 5).

```
K3Surface(x)
```

The K3 surface with the same data as that of the tuple x that is in the raw K3 database format.

117.10.2 Making New Databases

Here we explain in general terms how to write new databases of polarised varieties. We give complete instructions for writing the data and index files, but are more sketchy on the other steps: you will have to assemble the data, write translation functions and write any cosmetic wrapping functions for yourself.

Step 1: Prepare the directory. Make a new directory, 'NewDB' say, and copy the files `data_spec.m`, `write.m`, `init_info.m`, `write_tools.m`, `write_func.m`, `create_ind_func.m` into NewDB from the Magma package directories. Of these files, the first two will have to be edited to conform to the required database, while the remaining four are common to all databases.

Step 2: Decide record format. Edit the file `data_spec.m`. This determines which data is stored, and places unbreakable restrictions on that data. In particular, the choice of up to 5 indexing parameters is made here. Write intrinsics that translate between the data in its original representation and in its record representation.

Step 3: Update the writing functions. Edit the file `write.m`.

Step 4: Collect the data. Write all required data as records in one or more files. It is essential that the data is written to these files so that it can be read into a single Magma session in increasing order (with respect to the chosen indexes).

Step 5: Build the binary data and index files.

Step 6: Move the binaries to the data libraries.

Step 7: Final cosmetics. Write as Magma package code any cosmetic wrappers or search routines that are to be used with the new database. Document these intrinsics.

117.11 Bibliography

[**ABR02**] Selma Altinok, Gavin Brown, and Miles Reid. Fano 3-folds, $K3$ surfaces and graded rings. In *Topology and geometry: commemorating SISTAG*, volume 314 of *Contemp. Math.*, pages 25–53. Amer. Math. Soc., Providence, RI, 2002.

[**Alt98**] Selma Altinok. *Graded Rings Corresponding to Polarised K3 Surfaces and Q-Fano* 3*-folds.* PhD thesis, University of Warwick, 1998. URL:http://www.maths.warwick.ac.uk/~miles/students/Selma/.

[**Bro03**] Gavin Brown. Datagraphs in algebraic geometry. In *Symbolic and Numerical Scientific Computing – Proc. of SNSC'01*, volume 2630 of *LNCS*, Heidelberg, 2003. Springer-Verlag. F. Winkler and U. Langer (eds.).

[**Buc03**] Anita Buckley. *Orbifold Riemann-Roch for 3-folds and applications to Calabi-Yaus.* Phd thesis, Warwick University, 2003.

[**IF00**] Anthony Iano-Fletcher. Working with weighted complete intersections. In A. Corti and M. Reid, editors, *Explicit birational geometry of* 3*-folds*, pages 101 – 173. CUP, Cambridge, 2000.

[**KS00**] Maximilian Kreuzer and Harald Skarke. Complete classification of reflexive polyhedra in four dimensions. *Adv. Theor. Math. Phys.*, 4(6):1209–1230, 2000.

[**Mat89**] H. Matsumura. *Commutative ring theory.* CUP, ACUP, 1989.

[**Pap03**] S. Papadakis. Kustin–Miller unprojection with complexes. *J. Algebraic Geometry*, to appear, 2003.

[**Rei00**] Miles Reid. Graded rings and birational geometry. In K. Ohno, editor, *Proc. of algebraic geometry symposium (Kinosaki)*, pages 1 – 72. 2000. URL:http://www.maths.warwick.ac.uk/~miles/3folds/.

[**SB**] B. Szendrői and A. Buckley. Orbifold Riemann-Roch for threefolds with an application to Calabi-Yau geometry. math.AG/0309033.

[**Suz**] K. Suzuki. On Fano indices of Q-Fano 3-folds. Avaialable as math.AG/0210309 on arXiv.

118 TORIC VARIETIES

Chapter 118
TORIC VARIETIES

118.1 Introduction and First Examples

We describe a package to work quite generally with toric geometry within the scope of Magma's scheme machinery. From that point of view, we regard toric varieties as a vast array of possible ambient spaces for schemes, generalising the affine and projective spaces already available. But of course toric geometry is a large topic in its own right, and in the first place we present it for its own sake.

There are many different points of view on toric geometry. We model our approach on several sources: the primary ones are Danilov [Dan78] and Cox [Cox95], but also Fulton [Ful93] and Oda [Oda88].

One of the primary points of view, exemplified by Danilov [Dan78], is to regard the two-way relationship between a part of scheme theory and the combinatorics of polyhedra as foundational. This package can operate from that point of view.

Another, more recent, point of view is that of the Cox ring, introduced by Cox [Cox95], a (multi-)graded polynomial ring that works as a homogeneous coordinate ring for a toric variety (or indeed any variety with finitely-generated Picard group) in much the same way as the coordinate ring of affine or projective space. This package can also operate from this point of view, and indeed this is the central object in its design. Again, this is bound to impose practical restrictions on the dimensions in which calculations can reasonably be expected to work: we use these 'Cox coordinates' to define subschemes of toric varieties, which often leads to Gröbner basis calculations, and these are famously intolerant of the number of variables involved.

118.1.1 The Projective Plane as a Toric Variety

The projective plane is the toric variety corresponding to the fan with three maximal cones lying in the Euclidean plane:

$$\langle(0,1),(1,0)\rangle\,,\quad \langle(1,0),(-1,-1)\rangle\,,\quad \langle(-1,-1),(0,1)\rangle\,.$$

Example H118E1

We build the projective plane (defined over the rational field) as a toric variety. There are various simple constructors for this, but we do it slowly by constructing its fan first: we list the one-dimensional rays of the fan, and then describe each maximal cone by listing the sequence of indices of the rays that generate it.

```
> rays := [ [0,1], [1,0], [-1,-1] ];
> cones := [ [1,2], [1,3], [2,3] ];
> F := Fan(rays,cones);
> F;
```

```
Fan F with 3 rays:
    ( 0,  1),
    ( 1,  0),
    (-1, -1)
and 3 cones with indices:
    [ 1, 2 ],
    [ 1, 3 ],
    [ 2, 3 ]
```

This is enough to determine a toric variety; we name its natural (Cox) coordinates as x, y, z during the definition. (Since this is a scheme, it needs to have a base ring assigned.)

```
> X<x,y,z> := ToricVariety(Rationals(),F);
> X;
Toric variety of dimension 2
Variables: x, y, z
The irrelevant ideal is:
    (z, y, x)
The grading is:
    1, 1, 1
```

As with all schemes in Magma, points can be created by coercing sequences of coefficients (either from the given base field or from an extension of that) into the scheme.

```
> X ! [1,2,3];
(1 : 2 : 3)
```

When working over an extension k of the base field, one must coerce into the point set (of k-valued points of X).

```
> k<i> := QuadraticField(-1);
> X(k);
Set of points of X with coordinates in k
> X(k) ! [1,i,2*i];
(1 : i : 2*i)
```

The irrelevant ideal (described in the display of X above) describes the locus of coordinates that do not represent points; in this case, the irrelevant ideal records the fact that $(0 : 0 : 0)$ is not a point of projective space.

```
>  X ! [0,0,0];

>> X ! [0,0,0];
     ^
Runtime error in '!': Illegal coercion
> Y<u,v,w> := ProjectiveSpace(Rationals(),2);
> Y;
Projective Space of dimension 2
Variables: u, v, w
> IrrelevantIdeal(Y);
Ideal of Polynomial ring of rank 3 over Rational Field
```

```
Order: Graded Reverse Lexicographical
Variables: u, v, w
Homogeneous
Basis:
[
    u,
    v,
    w
]
> Gradings(Y);
[
    [ 1, 1, 1 ]
]
> Fan(Y);
Fan with 3 rays:
    ( 1,  0),
    ( 0,  1),
    (-1, -1)
and 3 cones with indices:
    [ 1, 2 ],
    [ 1, 3 ],
    [ 2, 3 ]
```

118.1.2 Resolution of a Nonprojective Toric Variety

Danilov translated the projectivity of a toric variety into properties of the fan. In geometry, projectivity is equivalent to the existence of an ample line bundle. In the combinatorics of the fan, the requirement is the existence of positive linear functions on each cone that agree on the boundaries and are strictly convex across boundaries. This is one small part of the theory of divisors on toric varieties that we cover in much greater detail in Section 118.6.

Example H118E2

We construct an example of a complete but non-projective toric variety following Danilov's original example of how a fan can fail to admit a strictly convex piece-wise linear support function. We will build the fan by hand, first specifying exactly the (one-dimensional) rays that it contains.

```
> rays := [ [0,0,1], [4,0,1], [0,4,1], [1,1,1], [2,1,1], [1,2,1], [-1,-1,-1] ];
```

Now we specify the top-dimensional cones that the fan contains. This is done by naming the rays that generate each cone from the list of rays above. So, for example, the sequence `[1,3,6]` below refers to the cone generated by the 1st, 3rd and 6th rays, that is by $[0,0,1]$, $[0,4,1]$ and $[1,2,1]$.

```
> cones := [ [1,3,6], [1,4,6], [1,2,4], [2,4,5], [2,3,5], [3,5,6], [4,5,6],
>            [1,3,7], [1,2,7], [2,3,7] ];
```

This is enough information to determine a fan. The command below to construct the fan F with this data takes a little time, since it checks that the maximal cones we nominated are indeed maximal and intersect correctly to lie in a fan.

```
> F := Fan(rays,cones);
```

Finally we move away from the lattice world to the geometry by creating the toric variety corresponding to the fan F.

```
> X := ToricVariety(Rationals(),F);
```

We can now ask questions of X as we would with any other variety or ambient space.

```
> Dimension(X);
3
> IsComplete(X);
true
> IsProjective(X);
false
```

Unfortunately our specimen is flawed: it has singularities.

```
> IsNonsingular(X);
false
> Y := Resolution(X);
> IsProjective(Y);
false
```

The last line takes a little time: Magma computes the ample cone of Y and then determines whether or not it is empty. In this case, Y remains non-projective, although there is no reason to expect that: Danilov's original example can be made projective by blowing up a line (or simply flopping a line).

118.1.3 The Cox Ring of a Toric Variety

The Cox ring of a toric variety is its natural homogeneous coordinate ring. It is a multigraded ring (with other data besides). It is often easier to construct the gradings for a Cox ring than it is to describe a fan.

A Cox ring requires four pieces of data:

- a polynomial ring R (a ring of coordinates for an affine space)
- a sequence B of 'irrelevant' ideals (the loci defined by these are discarded from the affine space)
- a sequence Z of sequences of integral weights for R; each element has length the number of indeterminates of R
- a sequence Q of sequences of quotient gradings for R.

Example H118E3

Cox's original construction determines a Cox ring from the fan of a toric variety, and then realises the toric variety as the quotient of the affine space (with the irrelevant locus discarded) by the action of a torus (determined by the sequences of weights). But it is also possible simply to specify this data independently of a fan.

```
> R<u,v,x,y,z> := PolynomialRing(Rationals(),5);
> irrel1 := ideal< R | u,v >;
> irrel2 := ideal< R | x,y,z >;
> B := [ irrel1, irrel2 ];
> Zwts := [
>             [ 1, 1, 0, -1, -3 ],
>             [ 0, 0, 1,  2,  3 ] ];
> Qwts := [];
> C := CoxRing(R,B,Zwts,Qwts);
> C;
Cox ring C with underlying Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: u, v, x, y, z
The components of the irrelevant ideal are:
    (z, y, x), (v, u)
The 2 gradings are:
    1,  1,  0, -1, -3,
    0,  0,  1,  2,  3
```

The pieces of data can be retrieved by `Gradings(C)`, `QuotientGradings(C)` and so on.
In favourable cases, a Cox ring does indeed arise from a fan using Cox's construction. This fan can be recovered.

```
> F := Fan(C);
> F;
Fan F with 5 rays:
    ( 1,  0,  0),
    ( 0,  1,  0),
    ( 1,  1,  3),
    (-2, -2, -3),
    ( 1,  1,  1)
and 6 cones with indices:
    [ 1, 3, 4 ],
    [ 1, 3, 5 ],
    [ 1, 4, 5 ],
    [ 2, 3, 4 ],
    [ 2, 3, 5 ],
    [ 2, 4, 5 ]
```

Alternatively, one can construct a toric variety from a Cox ring—as in Cox's construction, the Cox ring contains exactly the data required to construct a variety as a torus quotient.

```
> X := ToricVariety(C);
> Dimension(X);
```

```
3
```

Sequences of coefficients not lying in the locus of the irrelevant ideal can be interpreted as closed points of X as usual.

```
> X ! [1,0,1,0,0];
(1 : 0 : 1 : 0 : 0)
>  X ! [1,0,0,0,0];

>> X ! [1,0,0,0,0];
     ^
Runtime error in '!': Illegal coercion
```

The attempted coercion of the second point fails because its x, y and z coordinates (the 3rd, 4th and 5th coefficients of the vector) are all zero, but the locus $x = y = z = 0$ is defined by a component of the irrelevant ideal and so has been discarded: points of X do not have all three of those coordinates simultaneously equal to zero.

The homogeneous coordinates that the Cox ring provides for X behave in a very similar way to the homogeneous coordinates on projective space. In particular, one can define subschemes of X by the vanishing of polynomials in the Cox ring that are homogeneous with respect to all the **Z**-gradings (and the quotient gradings too).

```
> f := x^4*y + u^2*y^3 + v^5*z^2;
> Multidegree(X,f);
[ -1, 6 ]
[]
```

The multidegree is returned as two sequences: the first is the sequence of degrees of f with respect to each of the **Z**-gradings in turn, and the second is that with respect to the quotient gradings (in this case there are none). One defines a scheme as usual in Magma.

```
> V := Scheme(X, f);
> V;
Scheme over Rational Field defined by
u^2*y^3 + v^5*z^2 + x^4*y
> Dimension(V);
2
```

Much of Magma's scheme machinery works for schemes inside toric varieties, although at this stage some does not. (Some functions are missing because of the absence of standard nonsingular affine patches on toric varieties. In some cases these will be replaced by their orbifold analogues in due course.)

118.2 Fans in Toric Lattices

A *fan* F (in a toric lattice L) is a collection of cones in L satisfying typical conditions of cell decompositions (the inclusion of faces of cones is part of the data, and the requirement that any two cones of the fan intersect in a common face is enforced, for instance). The *support* of F is the union of all the cones as a subset of L. The main two cases are when either the support of F is the support of a single cone (which is regarded as being subdivided by the cones of the fan) or it is the whole of L. Other cases do occur and are allowed by our package—even having cones of different dimensions lying in complementary linear subspaces of L, although that is not commonly interpreted geometrically.

There are several constructors for well-known fans, and also standard methods for modifying fans. If these are not enough, then one can simply list the top-dimensional cones of a fan; Magma will check that they intersect correctly and will add the lower-dimensional cones as necessary.

Fans are of type `TorFan`.

118.2.1 Construction of Fans

We first list the comprehensive constructors for fans. After that, we have a collection of constructors for well-known fans, and there also are methods for modifying fans.

`Fan(Q)`

`define_fan`	BoolElt	*Default :* `false`
`max_cones`	BoolElt	*Default :* `false`

The fan generated by the cones in the sequence Q.

The optional parameter `define_fan` (by default `false`) can be set to `true`, so that the verification whether the input data is correct is skipped. The optional parameter `max_cones` (by default `false`) can be set to `true` to skip the verification whether the cones are maximal. Both will lead to errors later if the cones do not in fact determine a fan as claimed.

`Fan(R,S)`

`define_fan`	BoolElt	*Default :* `false`

The fan whose rays are the sequence of toric lattice points R (or sequences of sequences of integer coefficients of such points) and whose maximal cones correspond to the sequence S of sequences of integers: each such sequence is interpreted as the indices of a collection of rays of R, and these rays are used to generate a cone.

This constructor checks that the given data does indeed define a fan. This check can be lengthy. It can be omitted by setting the parameter `define_fan` to be `false`, although this should be used with extreme care: there is no telling what might go wrong if incorrect data is assumed to be a fan.

`Fan(C)`

The fan comprising all faces of the cone C.

`FanOfAffineSpace(n)`

The standard fan of affine space of dimension n, where n is a positive integer.

`FanOfWPS(W)`

A standard fan for the weighted projective space with weights W, a sequence of positive integers.

`#FanOfProjectiveSpace(n)`

The standard fan of projective space of dimension n, a positive integer.

`FanOfFakeProjectiveSpace(W,Q)`

A standard fan for the fake weighted projective space with weights W, a sequence of positive integers, and finite cyclic group actions given Q, a sequence of sequences of rational numbers. The finite cyclic actions are determined as follows: for $\mathbf{Z}/r$ to act diagonally with weights $(a_1, \ldots, a_n)$, include the sequence `[a1/r,...,an/r]` as an element of Q.

`ZeroFan(L)`

The fan in the toric lattice L supported at the origin.

`NormalFan(F,C)`

The normal fan to a cone C in a fan F in the toric lattice that is the quotient of the ambient lattice of F by the span of C; the quotient map of lattices is the second return value.

`SpanningFan(P)`

The toric fan that spans the polyhedron P; in particular, the rays of the fan are generated by the vertices of P.

`DualFan(P)`

The toric fan dual to the polyhedron P.

Example H118E4

The spanning fan of a polytope P is a fan in the same lattice as P with rays passing through its vertices. Here we make the fan of the Hirzebruch surface $\mathbf{F}_1$ as a spanning fan.

```
> P := Polytope([ [0,1], [1,0], [1,-1], [-1,0] ]);
> SpanningFan(P);
Fan with 4 rays:
    ( 0,  1),
    ( 1,  0),
    ( 1, -1),
    (-1,  0)
and 4 cones with indices:
    [ 1, 4 ],
    [ 1, 2 ],
    [ 2, 3 ],
    [ 3, 4 ]
```

The dual fan of a polytope P is a fan in the dual lattice to that of P whose rays are constant when evaluated on some facet of P. Here we make the fan of projective 3-space as a dual fan.

```
> Q := StandardSimplex(3);
> DualFan(Q);
Fan with 4 rays:
    (-1, -1, -1),
    ( 1,  0,  0),
    ( 0,  1,  0),
    ( 0,  0,  1)
and 4 cones with indices:
    [ 2, 3, 4 ],
    [ 1, 3, 4 ],
    [ 1, 2, 4 ],
    [ 1, 2, 3 ]
Mapping from: 3-dimensional toric lattice (Z^3)^* to 3-dimensional toric lattice
(Z^3)^* given by a rule
```

```
Blowup(F,v)
```

The blowup of the toric fan F at the point v that is an element of the ambient toric lattice of F.

```
IsInSupport(v,F)
```

Return `true` if and only if the support of the fan F contains the element v of its ambient toric lattice. In this case, the index of the first cone of F that contains v is also returned.

```
Fan(F1,F2)
F1 * F2
Fan(Q)
F ^ n
```

The nth Cartesian product of the fan F.

```
F eq G
```

Return `true` if and only if the two fans F and G are equal as objects in MAGMA (not simply isomorphic as fans).

118.2.2 Components of Fans

One can retrieve all the usual components of fans: their cones, rays and other dimensional skeletons, and so on.

Note the novelty of 'virtual' rays. Virtual rays occur when the rays of the fan in question do not span the ambient lattice, and so one may easily never encounter them. They occur if and only if the underlying toric variety is a product of $\mathbf{C}^*$ times X for a smaller toric variety X. If this the case, the fan is contained in a hyperplane (or smaller linear space, if there are more factors of $\mathbf{C}^*$). Virtual rays will be in the direction transversal to this hyperplane. Thus a virtual ray is not an honest ray, but a formal object introduced to allow $\mathbf{C}^*$ as a toric variety and give meaning to its coordinate. If virtual rays are not specified on creation, then Magma will decide where to put them (later, when it needs them).

```
Skeleton(F,n)
```

The fan generated by cones of the fan F of dimension at most the integer n.

```
C in F
```

Return `true` if and only if the cone C is one of the cones of the fan F.

```
Cones(F)
```

A sequence of the maximal cones of the fan F.

```
Cones(F,i)
```

The sequence of all i-dimensional cones in the fan F.

```
ConesOfCodimension(F,i)
```

The sequence of all codimension i cones in the fan F.

```
AllCones(F)
```

A sequence of all the cones of the fan F.

```
Cone(F,i)
```

The ith cone of the sequence of maximal cones of the fan F.

```
Cone(F,S)
```

The cone of the fan F spanned by the rays of indices given by the sequence S of integers. (If the optional parameter `extend` is set to `true`, then the smallest cone containing the given rays will be returned.)

```
SingularCones(F)
```

A sequence containing the minimal singular cones of the fan F. A second (parallel) sequence contains sets of the indices of the rays that generate these cones.

Example H118E5

The weighted projective space $\mathbf{P}(1, 2, 2, 3)$ has a singular line (with stabiliser $\mathbf{Z}/2$) and a singular point (with stabiliser $\mathbf{Z}/3$).

```
> F := FanOfWPS([1,2,2,3]);
> SingularCones(F);
[
    2-dimensional simplicial cone with 2 minimal generators:
        (-1, -1,  0),
        (-1,  1,  0),
    3-dimensional simplicial cone with 3 minimal generators:
        (-1, -1,  0),
        ( 1, -1, -1),
        ( 1,  0,  1)
]
[
    { 1, 4 },
    { 1, 2, 3 }
]
```

The two cones correspond to these two singular strata; the point strata lying on the line are not returned, since they can be recovered, if needed, as the cones having this 2-dimensional cone in their boundary.

`ConesOfMaximalDimension(F)`

A sequence of maximal cones of the fan F when ordered by inclusion. (This is the same as `Cones(F)` if the support of F is equidimensional.)

`ConeIndices(F)`

The sequence S of sets of integers, such that ith cone of the fan F is generated by rays with indices $S[i]$.

`ConeIndices(F,C)`

The sequence of integers that are the indices of the rays which generate the cone C of the fan F.

`ConeIntersection(F,C1,C2)`

The intersection of the two cones C_1 and C_2, both of which are members of the fan F. (This is usually more efficient than `C1 meet C2` given that the fan F exists.)

`Face(F,C)`

The smallest cone in the fan F which contains the cone C. (An error is returned if there is no such cone in the fan.)

`DualFaceInDualFan(P,Q)`

The cone in the toric fan dual to the polyhedron P which is dual to the face of P determined by the sequence of integers Q.

`Rays(F)`

A sequence containing the rays of the fan F (as a sequence of primitive lattice points on each ray).

`Ray(F,i)`

The ith ray of the fan F (regarded as the primitive ambient toric lattice point on the ray).

`AllRays(F)`

A sequence of the rays of the fan F (including all virtual rays).

`PureRays(F)`

A sequence of the (non-virtual) rays of the fan F.

`PureRayIndices(F)`

A sequence of the indices of the non-virtual rays of the fan F among all its rays.

`VirtualRays(F)`

A sequence of the virtual rays of the fan F.

`VirtualRayIndices(F)`

A sequence of the indices of the virtual rays of the fan F among all its rays.

118.2.3 Properties of Fans

`Ambient(F)`

The ambient toric lattice of the toric fan F.

`IsComplete(F)`

Return `true` if and only if the toric fan F has its entire ambient toric lattice; that is, the cones of F cover the whole ambient space.

`IsSingular(F)`

Return `false` if and only if all the cones of the fan F are nonsingular.

`IsNonsingular(F)`

Return `true` if and only if all the cones of the fan F are nonsingular.

`IsQFactorial(F)`

Return `true` if and only if all the cones of the fan F are **Q**-factorial.

`IsTerminal(F)`

Return `true` if and only if all the cones of the fan F are terminal.

`IsCanonical(F)`

Return `true` if and only if all the cones of the fan F are canonical.

`IsGorenstein(F)`

Return `true` if and only if all the cones of the fan F are Gorenstein.

`IsQGorenstein(F)`

Return `true` if and only if all the cones of the fan F are **Q**-Gorenstein.

118.2.4 Maps of Fans

`F @ f`

The image of the fan F by the map f of toric lattices.

`SimplicialSubdivision(F)`

`SimplicialSubdivision(C)`

A toric fan that is a simplicial subdivision of the toric fan F (or of the toric cone C).

Example H118E6

If C is a cone on a square, then it has two small simplicial subdivisions—the two sides of a standard flop, in fact. The simplicial subdivision intrinsic selects one of these.

```
> L := ToricLattice(3);
> C := Cone([L| [1,0,0], [0,1,0], [0,0,1],[1,-1,1]]);
> SiC := SimplicialSubdivision(C);
> #Cones(SiC);
2
> [ ZGenerators(B) : B in Cones(SiC) ];
[
    [
        (0, 1, 0),
        (1, -1, 1),
        (1, 0, 0)
    ],
    [
        (0, 1, 0),
        (1, -1, 1),
        (0, 0, 1)
    ]
]
```

`IsFanMap(F1,F2)`

Return `true` if and only if the two fans F_1 and F_2 lie in the same toric lattice, and each cone of F_1 is a subcone of some cone of F_2.

`IsFanMap(F1,F2,f)`

Return `true` if and only if the toric lattice map f between the ambient toric lattices of the two fans F_1 and F_2 maps every cone of F_1 into a cone of F_2.

`ResolveFanMap(F1,F2)`

A toric fan F that resolves the identity map of lattices restricted to the toric fans F_1 and F_2. The two fans F_i are expected to lie in the same lattice and to have the same support, and the resulting toric fan F gives a common refinement of them; in particular, F will admit a fan map into each of the F_i. (If the F_i have different supports, the fan F will be supported on their intersection and will only refine this part of each of them. Geometrically speaking, this will produce a non-proper resolution.)

118.3 Geometrical Properties of Cones and Polyhedra

`IsSingular(C)`

Return `true` if and only if the affine variety associated with the cone C is singular.

`IsNonsingular(C)`

Return `true` if and only if the affine variety associated with the cone C is nonsingular.

`IsSmooth(P)`

Return `true` if and only if the polyhedron P is a smooth polytope.

`IsGorenstein(C)`

Return `true` if and only if the cone C has (the primitive points on its) rays contained in an affine hyperplane that is defined by an integral equation.

`IsReflexive(P)`

Return `true` if and only if the polyhedron P is reflexive; i.e. P and its dual $P^\vee$ are both integral polytopes.

IsQGorenstein(C)

Return `true` if and only if the cone C has (the primitive points on its) rays contained in an affine hyperplane.

GorensteinIndex(C)

The Gorenstein index of the affine variety corresponding to the cone C together with the dual vector determining the equation of the hyperplane. (It is an error if C is not $\mathbf{Q}$-Gorenstein.)

GorensteinIndex(P)

The Gorenstein index of the lattice polytope P; i.e. the smallest positive integer k such that $kP^\vee$ is an integral polytope.

IsQFactorial(C)

IsSimplicial(P)

Return `true` if and only if the cone C or polytope P is simplicial.

IsTerminal(C)

Return `true` if and only if the singularity of the affine variety associated to the cone C is (at worst) terminal.

IsCanonical(C)

Return `true` if and only if the singularity of the affine variety associated to the cone C is (at worst) canonical.

IsFano(P)

Return `true` if and only if the polyhedron P is a Fano polytope (i.e. of maximum dimension in the ambient lattice, containing the origin strictly in its interior, with primitive lattice vertices).

Example H118E7

We make the cone corresponding to the (affine) terminal quotient singularity $\mathbf{C}^3/(\mathbf{Z}/5)$ where $\mathbf{Z}/5$ acts as the 5th roots of unity in the diagonal representation diag$(1, 2, 3)$.

```
> L := ToricLattice(3);
> v := L ! [1/5,2/5,3/5];
> LL,emb := AddVectorToLattice(v);
> C := PositiveQuadrant(L);
> CC := Image(emb,C);
> CC;
Cone CC with 3 generators:
    (1, 0, 0),
    (0, 1, 0),
    (3, 1, 5)
```

We can check that this really is terminal and compute its Gorenstein index, the least positive multiple of the canonical class that is Cartier.

```
> IsTerminal(CC);
true
> GorensteinIndex(CC);
5 (1, 1, -3/5)
```

We can compute a resolution of singularities of this cone, the analogue of a simplicial subdivision for cones, although we must treat it as a fan to do so.

```
> F := Fan(CC);
> F;
Fan F with 3 rays:
    (0, 1, 0),
    (1, 0, 0),
    (3, 1, 5)
and one cone with indices:
    [ 1, 2, 3 ]
> Resolution(F);
Fan with 8 rays:
    (0, 1, 0),
    (1, 0, 0),
    (3, 1, 5),
    (2, 1, 2),
    (1, 1, 1),
    (3, 1, 3),
    (3, 1, 4),
    (2, 1, 3)
and 11 cones
```

Note that this is not a minimal resolution: such a resolution would only need to subdivide at the four additional rays at the (original) lattice points $1/5(1,2,3)$, $1/5(2,4,1)$, $1/5(3,1,4)$ and $1/5(4,3,2)$.

118.4 Toric Varieties

118.4.1 Constructors for Toric Varieties

We list some simple constructors for simple toric varieties. There are more general constructors for toric varieties (either from their fans or their Cox rings) in other sections.

```
ToricVariety(k,n)
```

Projective n-space $\mathbf{P}^n$ defined over the field k as a toric variety.

```
ToricVariety(k,Z)
```

The (weighted) projective space $\mathbf{P}(Z)$ defined over the field k with weights the positive integer sequence Z as a toric variety.

```
ToricVariety(k,Z,Q)
```

The fake weighted projective space defined over the field k with weights the positive integer sequence Z and a single sequence of quotient weights the sequence Q of rational numbers.

```
ToricVariety(k,M,v)
```

The n-dimensional toric variety $n \geq 2$ defined over the field k with weights begin the two sequences of integers (of the same length $n+2$) that comprise M and linearisation the length 2 integer sequence v. (This toric variety is the GIT quotient of k^{n+2} by a 2-dimensional torus acting with weights M and linearisation v. To get a toric variety of the right dimension, v must lie in the mobile cone implicit in the notation. In practice, this means that the columns of M must generate a cone with vertex in a 2-dimensional toric lattice and v must lie in the 'very-interior' of that cone, in the sense that it must lie in the strict interior of C and in the subcone generated by all columns of M except the two most extreme.)

Example H118E8

We build a Hirzebruch surface as a GIT quotient.

```
> X<u,v,x,y> := ToricVariety(Rationals(),[[1,1,0,-1],[0,0,1,1]],[1,1]);
> X;
Toric variety of dimension 2
Variables: u, v, x, y
The components of the irrelevant ideal are:
    (y, x), (v, u)
The 2 gradings are:
    1,  1,  0, -1,
    0,  0,  1,  1
```

The polarisation $(1,1)$ that we used is forgotten—all that is left is X.

```
ToricVariety(k)
```

The zero-dimensional point over the field k defined as a toric variety.

```
ProjectiveSpace(k,n)
```

Projective n-space $\mathbf{P}^n$ defined over the field k.

```
ProjectiveSpace(k,W)
FakeProjectiveSpace(k,W,Q)
```

The (fake) weighted projective space over the field k with weights the sequence of integers W (and quotient weights the sequence of sequences of rational numbers, if provided).

118.4.2 Toric Varieties and Their Fans

```
ToricVariety(k,F)
```

The toric variety (defined over the field k) corresponding to the toric fan F.

```
Fan(X)
```

The toric fan corresponding to the toric variety X.

```
Rays(X)
```

The rays of the fan of the toric variety X.

```
OneParameterSubgroupsLattice(X)
```

The lattice of weights of the toric variety X; this is the lattice which supports the toric fan of X.

```
MonomialLattice(X)
```

The monomial lattice of the toric variety X, namely the toric lattice dual to that containing the fan of X.

```
CoxMonomialLattice(X)
```

The lattice whose elements represent Weil divisors on the toric variety X; it is dual to ray lattice of X.

```
DivisorClassLattice(X)
```

The divisor class lattice of the toric variety X.

```
IrrelevantIdeal(X)
```

A sequence of ideals that are the components of the irrelevant ideal of the toric variety X.

```
QuotientGradings(X)
```

A sequence of sequences of rational numbers describing the quotients by finite cyclic groups that arise in the construction of the toric variety X.

```
NumberOfQuotientGradings(X)
```

The number of sequences the generate the quotient gradings of the toric variety X.

118.4.3 Properties of Toric Varieties

```
IsSingular(X)
```

Return `false` if and only if the toric variety X is nonsingular.

```
IsNonsingular(X)
```

Return `true` if and only if the toric variety X is nonsingular.

```
IsGorenstein(X)
```

Return `true` if and only if the toric variety X is Gorenstein.

```
IsQGorenstein(X)
```

Return `true` if and only if the toric variety X is $\mathbf{Q}$-Gorenstein.

```
IsQFactorial(X)
```

Return `true` if and only if the toric variety X is $\mathbf{Q}$-factorial.

```
IsTerminal(X)
```

Return `true` if and only if the toric variety X has (at worst) terminal singularities.

```
IsCanonical(X)
```

Return `true` if and only if the toric variety X has (at worst) canonical singularities.

```
IsComplete(X)
```

Return `true` if and only if the toric variety X is complete.

```
IsProjective(X)
```

Return `true` if and only if the toric variety X is projective.

```
IsFano(X)
```

Return `true` if and only if the anticanonical divisor of the toric variety X is ample.

```
IsFakeWeightedProjectiveSpace(X)
```

Return `true` if and only if the toric variety X has exactly one $\mathbf{Z}$-grading.

```
IsWeightedProjectiveSpace(X)
```

Return `true` if and only if the toric variety X has exactly one $\mathbf{Z}$-grading and no quotient gradings.

118.4.4 Affine Patches on Toric Varieties

```
ToricAffinePatch(X,i)
```

The affine patch corresponding to i-th cone of fan of the toric variety X together with the inclusion map.

```
ToricAffinePatch(X,S)
```

```
ToricAffinePatch(X,S)
```

The toric variety, obtained from the toric variety X by set the monomials of the sequence S set to be non-zero (or alternatively the variables of X with indices from the sequence of integers S set non-zero). The inclusion map is returned as a second value.

118.5 Cox Rings

The Cox ring of a toric variety X is a polynomial ring whose variables are in bijection with the 1-skeleton of the fan of X together with three sequences of additional data:

(1) a sequence of the components of an ideal called *the irrelevant ideal*;

(2) a sequence of integral lattice points determining weights of $\mathbf{G}_m$ actions, called the **Z** *weights*;

(3) a sequence of rational lattice points determining weights of finite cyclic group actions, called the *quotient weights*.

When the Cox ring of a toric variety is displayed in Magma, all nontrivial data is also printed, but any sequences that are empty are omitted.

Cox rings provide a powerful way to construct toric varieties: under some mild conditions, specification of data of this nature determines a toric variety.

118.5.1 The Cox Ring of a Toric Variety

Cox [Cox95] associates a ring, now called the *Cox ring*, to a toric variety, and Magma allows exactly the same construction.

```
CoxRing(X)
```

The Cox ring of the toric variety X.

```
CoxRing(k,F)
```

The Cox ring of the toric variety defined over field k by the fan F.

Example H118E9

We build the weighted projective space $\mathbf{P}^2(1,2,3)$.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),[1,2,3]);
```

The Cox ring of $\mathbf{P}^2(1,2,3)$ is the usual homogeneous coordinate ring, graded by the weights $1,2,3$ of the space—that is, x has weight 1, y has weight 2 and z has weight 3.

```
> CoxRing(P);
Cox ring with underlying Graded Polynomial ring of rank 3 over Rational Field
Order: Graded Reverse Lexicographical
Variables: x, y, z
Variable weights: [1, 2, 3]
The irrelevant ideal is:
    (x, y, z)
The grading is:
    1, 2, 3
```

The irrelevant ideal is the usual one for projective spaces: it decrees that $(0,0,0)$ is not a point of $\mathbf{P}^2$ since it lies in the locus defined by the irrelevant ideal.

Example H118E10

We build a toric variety X_2 whose fan resembles that of $\mathbf{P}^2$.

```
> F2 := Fan([[1,2],[-2,-1],[1,-1]], [[1,2],[1,3],[2,3]]);
> X2<u,v,w> := ToricVariety(Rationals(),F2);
```

However, X is not isomorphic to $\mathbf{P}^2$. The small catch is that the 1-skeleton of the fan F_2 that we defined (in other words, those three vectors $(1,2)$, $(-2,-1)$ and $(1,-1)$) does not generate the lattice N, but only a sublattice. So X will be the quotient of $\mathbf{P}^2$ by some finite group action.

```
> CoxRing(X2);
Cox ring with underlying Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: u, v, w
The irrelevant ideal is:
    (w, v, u)
The quotient grading is:
    1/3( 0, 2, 1 )
The integer grading is:
         1, 1, 1
```

The returned data are very similar to those for the Cox ring of $\mathbf{P}^2$. The difference is in the third piece of data: a sequence containing the single element $1/3(0,2,1)$. This indicates that X is the quotient of $\mathbf{P}^2$ by the action of $\mathbf{Z}/3$ given by

$$\varepsilon : (u,v,w) \mapsto (u, \varepsilon^2 v, \varepsilon w).$$

where ε is a cube-root of unity.

118.5.2 Cox Rings in Their Own Right

The introduction to this section describes Cox rings in abstract terms. It is possible to define them as polynomial rings plus additional data without naming a toric variety or a fan in the first place.

```
CoxRing(R,B,Z,Q)
```

The Cox ring with polynomial ring R of rank n (that is, having n variables) and additional data as follows: B is a sequence of ideals (or of sequences of elements of R, each of which will be interpreted as the generators of ideals); Z is a sequence of sequences of integers, each one of length n; Q is a sequence of sequences of rationals, each one of length n.

The sequence B is regarded as the components of the irrelevant ideal, and Z and Q are the $\mathbf{Z}$ weights and quotient weights respectively.

```
C1 eq C2
```

Return `true` if and only if the two Cox rings C_1 and C_2 have the same underlying polynomial rings and are defined by the same combinatorial data.

```
BaseRing(C)
```

```
CoefficientRing(C)
```

The coefficient field of the Cox ring C.

```
UnderlyingRing(C)
```

The underlying polynomial ring of the Cox ring C.

```
Length(C)
```

The rank of the underlying polynomial ring of the Cox ring C, that is, the number of polynomial variables of C.

```
IrrelevantIdeal(C)
```

The irrelevant ideal of the Cox ring C.

```
IrrelevantComponents(C)
```

A sequence containing the components of the irrelevant ideal of the Cox ring C.

```
IrrelevantGenerators(C)
```

A sequence of sequences, each containing the generators of the components of the irrelevant ideal of the Cox ring C.

```
Gradings(C)
```

The $\mathbf{Z}$ gradings of the Cox ring C, that is, a sequence of sequences of integers.

```
NumberOfGradings(C)
```

The number of **Z** gradings of the Cox ring C.

```
QuotientGradings(C)
```

The quotient gradings of the Cox ring C, that is, a sequence of sequences of rational numbers.

```
NumberOfQuotientGradings(C)
```

The number of quotient gradings of the Cox ring C.

```
C . i
```

The i-th indeterminate for the underlying polynomial ring of the Cox ring C.

```
AssignNames(~C, S)
```

Procedure to change the printed names of the indeterminates of the Cox ring C. The ith indeterminate will be given name the ith element of the sequence S of strings (which has length at most the number of indeterminates of C). This does not change the names of the indeterminates for calling—this must be done with an explicit assignment or with the angle bracket notation when defining the Cox ring in the first place.

```
Name(C,i)
```

The ith variable of the underlying polynomial ring of the Cox ring C.

118.5.3 Recovering a Toric Variety From a Cox Ring

It is simple either to recover a toric variety from a Cox ring (if one exists at all) or the fan and associated lattice machinery corresponding to the toric variety. The algorithm is straightforward: use the **Z** weights to determine the rays of a fan and then the irrelevant ideal to construct the rest of the cone structure of the fan. The quotient weights may then require the lattice to be extended to a larger lattice but containing the same fan. With this point of view, the Cox ring can be regarded as the primary collection of data of a toric variety.

```
ToricVariety(C)
```

The toric variety whose Cox ring is C. It is not checked whether the Cox data defines a toric variety; if you are unsure, you should ask for the fan.

Example H118E11

Sometimes, rather than defining a fan, it is easier to construct a Cox ring first and build a toric variety from that.

```
> R<x1,x2,x3,y1,y2,y3,y4> := PolynomialRing(Rationals(),7);
> I := [ ideal<R|x1,x2,x3>, ideal<R|y1,y2,y3,y4> ];
> Z := [ [1,1,1,0,-3,-5,-5], [0,0,0,1,1,1,1] ];
> Q := [];
> C := CoxRing(R,I,Z,Q);
> C;
Cox ring C with underlying Polynomial ring of rank 7 over Rational Field
Order: Lexicographical
Variables: x1, x2, x3, y1, y2, y3, y4
The components of the irrelevant ideal are:
    (y4, y3, y2, y1), (x3, x2, x1)
The 2 gradings are:
    1,  1,  1,  0, -3, -5, -5,
    0,  0,  0,  1,  1,  1,  1
> X := ToricVariety(C);
> X;
Toric variety of dimension 5
Variables: x1, x2, x3, y1, y2, y3, y4
The components of the irrelevant ideal are:
    (y4, y3, y2, y1), (x3, x2, x1)
The 2 gradings are:
    1,  1,  1,  0, -3, -5, -5,
    0,  0,  0,  1,  1,  1,  1
```

Now Magma can compute the fan of X if we really want it.

```
> Fan(X);
Fan with 7 rays:
    ( 1,  0,  0,  0,  0),
    ( 0,  1,  0,  0,  0),
    ( 0,  0,  1,  0,  0),
    ( 1,  1,  1,  2,  0),
    (-3, -3, -3, -5,  0),
    ( 0,  0,  0,  0,  1),
    ( 2,  2,  2,  3, -1)
and 12 cones
```

`Fan(C)`

The fan associated to the Cox ring is C; an error is reported if there is no such fan.

`CoxMonomialLattice(C)`

The Cox monomial lattice of the Cox ring C.

```
DivisorClassLattice(C)
```

The divisor class lattice of the Cox ring C.

```
MonomialLattice(C)
```

The monomial lattice of the Cox ring C.

```
OneParameterSubgroupsLattice(C)
```

The one-parameter subgroups lattice of the Cox ring C.

```
RayLattice(C)
```

The ray lattice of the Cox ring C.

```
DivisorClassGroup(C)
```

The divisor class group of the Cox ring C.

```
RayLatticeMap(C)
```

The map from the ray lattice of the Cox ring C to the ambient lattice of its fan.

```
WeilToClassGroupsMap(C)
WeilToClassLatticesMap(C)
```

Comparison maps between toric lattices related to the Cox ring C of a toric variety.

118.6 Invariant Divisors and Riemann-Roch Spaces

Divisors on toric varieties work in the same way as on any other varieties, except that within each linear equivalence class it is possible to choose torus invariant representatives. These invariant divisors are composed of toric strata, and so in raw combinatorial terms one can regard divisors as being integer (or rational) labels on the rays of the fan. This is a convenient way to construct divisors, but there are many other methods.

As for other schemes on which divisor calculations are defined in Magma, divisors on a toric variety have a single divisor group as their parent. Divisors can be constructed by coercing appropriate data into this group, but this is not the only method so it can be ignored for most purposes. (This group is, however, the connection between divisors and the toric variety, so it is always alive in the background.)

118.6.1 Divisor Group

```
DivisorGroup(X)
```

The divisor group of the toric variety X. This is simply a parent object for divisors on X, and it is not computed as an abstract group.

```
ToricVariety(G)
```

The toric variety of which G is the divisor group.

```
G1 eq G2
```

Return `true` if and only if the divisor groups G_1 and G_2 are those of the same toric variety.

```
Divisor(G,S)
```

```
Divisor(G,S)
```

The divisor on the toric variety X associated to the divisor group G with coefficients given by the sequence S of integers or rationals with respect to the rays of the fan of X.

```
Divisor(G,i)
```

The divisor on the toric variety X associated to the divisor group G given by the vanishing of the ith coordinate of X.

118.6.2 Constructing Invariant Divisors

```
Divisor(X,S)
```

The Weil divisor (respectively $\mathbf{Q}$-Weil divisor) on the toric variety X whose multiplicity on the ith coordinate divisor is the ith element of the sequence S of integers (respectively, rational numbers).

```
Divisor(X,i)
```

The divisor on the toric variety X given by the vanishing of the ith coordinate of X.

```
Divisor(X,f)
```

The divisor on the toric variety X defined by the polynomial f of the Cox ring of X.

```
Divisor(X,m)
```

If m is in the monomial lattice of the toric variety X, this gives the principal divisor on X corresponding to the monomial m. If m is a form on the ray lattice of X, then this gives the Weil divisor corresponding to m.

`ZeroDivisor(X)`

The zero divisor on the toric variety X.

`Representative(X,m)`

`effective` BoolElt *Default :* `true`

A divisor D on the toric variety X whose class modulo linear equivalence equals m, an element of the divisor class group of X. Unless the parameter `effective` is set to `false`, D will be chosen to be effective if possible.

`Representative(X,m)`

`effective` BoolElt *Default :* `true`

A divisor D on the toric variety X whose class modulo linear equivalence equals m, an element of the Picard lattice or divisor class lattice of X. Unless the parameter `effective` is set to `false`, D will be chosen to be effective if possible.

`CanonicalDivisor(X)`

The canonical divisor of the toric variety X.

`CanonicalClass(X)`

`group` MonStgElt *Default :* "*Pic*"

The class of canonical divisor of the toric variety X. By default this is returned as an element of the Picard lattice of X (and X must be $\mathbf{Q}$-Gorenstein for this to make sense). However, the parameter `group` can be changed to `Cl` to return the divisor in the divisor class lattice or `ClZ` to return the divisor in the divisor class group.

`D1 + D2` `n * D` `- D` `D1 - D2` `D * v`

Standard arithmetic operations for divisors D, D_1, D_2 on a toric variety, where $n \in \mathbf{Q}$ and v is a point of the ambient toric lattice of the corresponding fan.

Example H118E12

We compute the Kawamata blowup of $1/7(1,2,5)$. First we construct the singular cone by hand:

```
> L := ToricLattice(3);
> C := PositiveQuadrant(L);
> v := L![1/7,2/7,5/7];
> LL,phi := AddVectorToLattice(v);
> CC := Cone(phi(Rays(C)));
> CC;
3-dimensional simplicial cone CC with 3 minimal generators:
    (1, 0, 0),
    (0, 1, 0),
    (4, 1, 7)
```

Now we compute the blowup.

```
> FF := Fan(CC);
> vv := phi(v);
> vv;
(3, 1, 5)
> GG := Blowup(Fan(CC),vv);
```

The blowup map is easy to recover:

```
> X := ToricVariety(Rationals(),FF);
> Y<x,y,z,t> := ToricVariety(Rationals(),GG);
> f := ToricVarietyMap(Y,X);
> f;
A map between toric varieties described by:
    (t)^(2/7)*(x),
    (t)^(1/7)*(y),
    (t)^(5/7)*(z)
```

Finally we shall compute the discrepancy of this Kawamata blowup. It should be 1/7.

```
> KX := CanonicalDivisor(X);
> KY := CanonicalDivisor(Y);
> KY - Pullback(f,KX);
Q-Weil divisor with coefficients:
    0, 0, 0, 1/7
```

118.6.3 Properties of Divisors

`Variety(D)`

The toric variety on which the divisor D is defined.

`Parent(D)`

The divisor group of a toric variety in which the divisor D lies.

`Weil(D)`

The multiplicities on rays of the fan of X that determine the invariant divisor D, where X is the toric variety on which D is defined.

`Cartier(D)`

The sequence of toric lattice elements (of the monomial lattice of X) that determine the divisor D on the toric affine patches of the toric variety X on which D lies. This requires that D be **Q**-Cartier.

`IsQCartier(D)`

Return `true` if and only if some integer multiple of the divisor D on a toric variety is Cartier.

`IsCartier(D)`

Return `true` if and only if the divisor D on a toric variety is Cartier.

`IsWeil(D)`

Return `true` if and only if the divisor D on a toric variety is a Weil divisor (that is, its coefficients are integers rather than rational numbers).

`IsAmple(D)`

Returns `true` if and only if the divisor D on a toric variety is ample.

`IsNef(D)`

Return `true` if and only if the divisor D on a toric variety is nef.

`IsBig(D)`

Return `true` if and only if the divisor D on a toric variety is big.

`PicardClass(D)`

The class in the Picard lattice corresponding to the **Q**-Cartier divisor D.

`MovablePart(D)`

The movable part (or mobile part) of the divisor D on a toric variety.

Example H118E13

We compute a variety as a blowup of the projective plane.

```
> X := ProjectiveSpace(Rationals(),[1,1,1]);
> Y<u,v,x,y> := Blowup(X, &+Rays(Fan(X))[1..2]);
> Y;
Toric variety of dimension 2
Variables: u, v, x, y
The components of the irrelevant ideal are:
    (y, x), (v, u)
The 2 gradings are:
    0, 0, 1, 1,
    1, 1, 1, 0
```

We consider a (toric coordinate) divisor on Y.

```
> D := Divisor(Y,4);
> MovablePart(D);
Weil divisor with coefficients:
    0, 0, 0, 0
> MovablePart(D) eq ZeroDivisor(Y);
```

```
true
```

The movable part of this divisor is the zero divisor. Adding a little bit of another effective divisor doesn't yet make a mobile divisor, but it has made it stably mobile: some multiple now has a movable part.

```
> E := D + (1/2)*Divisor(Y,u);
> MovablePart(E);
Weil divisor with coefficients:
    0, 0, 0, 0
>   MovablePart(2*E);
Weil divisor with coefficients:
    1, 0, 0, 1
>   MovablePart(2*E) eq (D + Divisor(Y,u));
true
```

`ImageFan(D)`

The dual fan to the rational polyhedron of sections of the divisor D on a toric variety X. If X is a complete variety, this will give the fan of Proj of the ring of sections of positive powers of D.

`Proj(D)`

Proj (as a toric variety) of the ring of sections of the divisor D on a toric variety. The map of underlying lattices which determines the map `Variety(D)` → `Proj(D)` is also returned.

`RelativeProj(D)`

The relative (sheaf) Proj of sections of the divisor D on a toric variety. If D is **Q**-Cartier, then the identity will be constructed; for non **Q**-Cartier divisors, a partial **Q**-factorialisation will be given.

`IntersectionForm(X,C)`

If the cone C is a codimension 1 face of the fan of the toric variety X, return the dual toric lattice vector that represents intersection of the toric subvariety corresponding to C.

118.6.4 Linear Equivalence of Divisors

```
IsQPrincipal(D)
```

Return `true` if and only if some integer multiple of the divisor D on a toric variety is principal.

```
IsPrincipal(D)
```

Return `true` if and only if the divisor D on a toric variety is principal.

```
IsLinearlyEquivalentToCartier(D)
```

Return `true` if and only if the divisor D on a toric variety is linearly equivalent to a Cartier divisor; if so, then a representative Cartier divisor is also returned.

```
AreLinearlyEquivalent(D,E)
IsLinearlyEquivalent(D,E)
```

Return `true` if and only if the divisors D and E are linearly equivalent.

```
DefiningMonomial(D)
```

The monomial (if D is effective) or rational monomial defining the divisor D on a toric variety.

```
LatticeElementToMonomial(D,v)
```

The monomial in the Cox ring that corresponds to the monomial lattice element v when regarded as a section of the divisor D.

118.6.5 Riemann–Roch Spaces of Invariant Divisors

```
RiemannRochPolytope(D)
```

The Riemann–Roch space of the divisor D as a polytope in the monomial lattice of the underlying toric variety.

```
RiemannRochBasis(D)
```

A basis of the Riemann–Roch space of the divisor D on a toric variety X: this is a sequence of rational functions on X.

```
RiemannRochDimension(D)
```

The dimension of the Riemann–Roch space of the divisor D on a toric variety.

```
GradedCone(D)
```

The graded cone of sections of multiples of the divisor D on a toric variety. In other words, the integral points of the ith graded piece of this cone represent sections of the divisor $i * D$.

```
Polyhedron(D)
```

The integral polyhedron whose integral points corresponds to sections of the divisor D.

Example H118E14

We make a simple toric variety Y by blowing up the plane.

```
> X := ProjectiveSpace(Rationals(),[1,1,1]);
> Y<u,v,x,y> := Blowup(X, &+Rays(Fan(X))[1..2]);
> Y;
Toric variety of dimension 2
Variables: u, v, x, y
The components of the irrelevant ideal are:
    (y, x), (v, u)
The 2 gradings are:
    0, 0, 1, 1,
    1, 1, 1, 0
```

We make a non-effective divisor as a difference of fibres of the natural map from Y to the line.

```
> D := 2*Divisor(Y,u) - Divisor(Y,v);
> IsEffective(D);
false
> P := Polyhedron(D);
> monos := [ LatticeElementToMonomial(D,v) : v in Points(P) ];
> monos;
[
    u,
    v
]
```

The polyhedron P is not quite the Riemann–Roch space of D, but it is for a divisor linearly equivalent to D.

```
> [ AreLinearlyEquivalent(Divisor(Y,m),D) : m in monos ];
[ true, true ]
```

`HilbertSeries(D)`

The Hilbert series of the divisor D on a toric variety X, namely

$$\Sigma_{m\geq 0} \dim H^0(X, mD).$$

This assumes that the spaces of sections $H^0(X, D)$ of D finite dimensional. This will be true if X is projective, for example, but it holds in other cases too.

`HilbertPolynomial(D)`

The Hilbert (quasi-)polynomial for the divisor D. The space of sections of D must be finite dimensional. That is, a sequence of polynomials $[p_0, \ldots, p_{r-1}]$ of length r, the quasi-period of the Hilbert polynomial, so that $\dim H^0(X, mD)$ is the value of $p_s(k)$ where $m = kr + s$ is the Euclidean division of m by r; in other words, s is the least residue of m modulo r. Note that since Magma indexes sequences from 1, we have that $p_i =$ `HilbertPolynomial(D)[i+1]`.

```
HilbertCoefficients(D,l)
```

The first $l + 1$ coefficients of the Hilbert series of the divisor D on a toric variety (starting with $0D$ up to and including lD).

```
HilbertCoefficient(D,i)
```

Thefirst ith coefficient of the Hilbert series of the divisor D on a toric variety.

Example H118E15

One can recreate the Cech cohomology calculation of the Riemann–Roch space of a divisor D on a toric variety X:

$$H^0(X, \mathcal{O}_X(D)) = \cap_\sigma(m_\sigma + \check{\sigma})$$

where the sum is taken over top-dimensional cones σ in the fan of X, and m_σ is the monomial in the dual lattice which determines D on the affine patch X_σ; $\check{\sigma}$ is the cone dual to σ.

```
> X<x,y,z> := ProjectiveSpace(Rationals(),[1,3,5]);
> D := Divisor(X,[2,3,1]);
> cones := Cones(Fan(X));
> RRD := &meet [ Polytope([Cartier(D)[i]]) + Dual(cones[i]) : i in [1..3]];
> IsPolytope(RRD);
true
```

We could compute the number of points of this cone RRD by saying `#Points(RRD)`. This should be regarded as a slow method to determine the number of points—after all, it requires us to find all the points before counting them. There is a specialised point-counting intrinsic `NumberOfPoints` which does not find the points first. The latter should be used for point counting, although in relatively small examples the former can be a little faster.

```
> NumberOfPoints(RRD);
14
```

To compute also the number of points of integral dilations of this polytope, we revert to using the divisor and computing its Hilbert series (or a few coefficients if that's all we need).

```
> time HilbertCoefficients(D,10);
[ 1, 14, 44, 92, 156, 238, 337, 452, 585, 735, 902 ]
> h<t> := HilbertSeries(D);
> h;
(-4*t^8 - 21*t^7 - 38*t^6 - 51*t^5 - 51*t^4 - 47*t^3 - 30*t^2 - 13*t - 1)/(t^9 -
    t^8 - t^6 + t^5 - t^4 + t^3 + t - 1)
> h * (1 - t) * (1 - t^3) * (1 - t^5);
4*t^8 + 21*t^7 + 38*t^6 + 51*t^5 + 51*t^4 + 47*t^3 + 30*t^2 + 13*t + 1
```

118.7 Maps of Toric Varieties

The initial method of expressing some maps between toric varieties is to derive them from maps between their associated lattices. These can also then be presented in terms of the variables of the Cox ring; this is the usual method for describing toric maps between projective spaces, for instance. This often results in radical expressions such as

$$(u, v) \mapsto (\sqrt{u}, v, v\sqrt{u}).$$

This example could be describing a map from $\mathbf{P}^1$ to $\mathbf{P}(1, 2, 3)$, for example: it is a 'monomial' map which observes the gradings. In that case, we could represent the same map by $(u, v) \mapsto (u, uv, u^2v)$ or $(u, v) \mapsto (1, v/u, v/u)$, or a host of other expressions. These two expressions have the benefit that they are polynomial or rational functions in u and v, and so they automatically define rational maps, but they have disadvantages too: for example, evaluating the map at the point $(0, 1)$ is not defined for these expressions, whereas it gives image $(1, 0, 1)$ in the original radical expression. (Notice that the choice of root does not matter, as long as it is assumed that the same choice $a = \sqrt{u}$ is made at each coordinate.)

More generally, one can define all maps between toric varieties (not just those arising from maps of lattices) using an appropriate notion of 'rational radical function', defined in terms of the polynomial Cox coordinates. This is very common when describing maps between standard projective spaces: one writes down a sequence of homogeneous polynomials of the same degree, without demanding that they are monomials.

The key point is that a rational map between varieties pulls rational functions (that are defined on the image) back to rational functions. It is enough to test this on a basis of rational functions. In the example above, if x, y, z are the coordinates on $\mathbf{P}(1, 2, 3)$ then y/x^2 and z/x^3 form a basis, and these both pull back to v/u, which is a rational function on $\mathbf{P}^1$.

We allow maps to be constructed from maps of the underlying toric lattices of fans. When displayed, they are described in these radical polynomial terms.

118.7.1 Maps from Lattice Maps

```
ToricVarietyMap(X,Y,f)
```

```
ToricVarietyMap(X,Y)
```

The rational map between toric varieties X and Y determined by the map f between their respective toric lattices (that is, the lattices underlying their respective fans). If the map f is not specified, it is assumed to be the identity map (and X and Y are assumed to have the same toric lattice).

```
Blowup(X,v)
```

The blowup of the toric variety X at the toric lattice point v of the toric lattice containing the fan of X; the natural map from the blowup to X is also returned.

```
IdentityMap(X)
```

The identity map on the toric variety X.

118.7.2 Properties of Toric Maps

```
IsRegular(f)
```

Return `true` if and only if the map f between toric varieties is regular.

```
IndeterminacyLocus(f)
```

A sequence of subschemes of the toric variety that is the domain of the map f between toric varieties at which f is not defined. (Note that these subschemes may in fact be empty.)

Example H118E16

We build a map from a Hirzebruch surface using the complete linear system of divisor.

```
> F2<u,v,x,y> := HirzebruchSurface(Rationals(),2);
> D := Divisor(F2,x);
> Y,f := Proj(D);
> Y;
Toric variety of dimension 2
Variables: $.1, $.2, $.3
The irrelevant ideal is:
    ($.3, $.2, $.1)
The grading is:
    1, 1, 2
> f;
Mapping from: 2-dimensional toric lattice N to 2-dimensional toric lattice N
given by a rule
```

The image variety Y is clearly the weighted projective space $\mathbf{P}(1,1,2)$. The map f returned is a map of underlying lattices. We can convert it into a map of the toric varieties, after which it will be presented in Cox coordinates.

```
> F := ToricVarietyMap(F2,Y,f);
> F;
A map between toric varieties described by:
    1,
    (v)*(u)^(-1),
    (x)*(y)^(-1)*(u)^(-2)
```

Now we can ask whether this map F is a morphism.

```
> IsRegular(F);
true
```

118.8 The Geometry of Toric Varieties

118.8.1 Resolution of Singularities and Linear Systems

```
Resolution(X)
```

A resolution of singularities of the toric variety X together with the natural morphism from the resolution to X. The resolution is not necessarily minimal.

```
ResolveLinearSystem(D)
```

The toric variety Y whose fan lives in the same lattice as the fan of the toric variety X on which the divisor D is defined, such that Y resolves the map given by the linear system of D.

118.8.2 Mori Theory of Toric Varieties

```
MoriCone(X)
```

The Mori Cone of toric variety X (as an abstract cone), that is, the cone generated by numerical classes of torus invariant curves on X.

```
NefCone(X)
```

The nef Cone of toric variety X (as an abstract cone).

```
ExtremalRays(X)
```

```
ExtremalRayContractions(X)
```

The images of extremal contractions of rays in the nef-cone of the toric variety X.

```
ExtremalRayContraction(X,i)
```

The toric variety that is the image of the ith extremal contraction of the toric variety X; the contraction morphism is returned as a second value.

```
ExtremalRayContractionDivisor(X,i)
```

The toric divisor that gives the ith extremal contraction of the toric variety X (that is, the divisor is the pullback of an ample divisor on the image).

```
TypeOfContraction(X,i)
```

```
TypesOfContractions(X)
```

A string describing the ith extremal contraction of the toric variety X (or all together in a sequence if i is not specified).

```
IsMoriFibreSpace(X,i)
```

Return `true` if and only if the ith extremal ray of the toric variety X gives an extremal contraction to a variety of lower dimension than X.

`IsDivisiorialContraction(X,i)`

Return `true` if and only if the ith extremal ray of the toric variety X gives a divisorial contraction of X.

`IsFlipping(X,i)`

Return `true` if and only if the ith extremal ray of the toric variety X gives a small contraction of X; in this case the intrinsic `Flip(X,i)` provides the flipped toric variety.

`Flip(X,i)`

The flipped variety of the ith extremal contraction of the toric variety X, assuming that this extremal contraction is of flipping type.

`Flip(D)`

The (generalised) flip of the morphism given by the $\mathbf{Q}$-Cartier divisor, assuming that this morphism is small.

`WeightsOfFlip(X,i)`

The weights of a $\mathbf{G}_{\mathrm{m}}$ action whose variation would give the flip of the ith extremal contraction of the toric variety X, assuming that this extremal contraction is of flipping type.

Example H118E17

```
> F0 := FanOfWPS([1,1,1,1]);
> L3 := Ambient(F0);
> F := Blowup(F0,L3 ! [2,-5,3]);
> X := ToricVariety(Rationals(),F);
> ExtremalRays(X);
[
    (0, -1),
    (1, 56)
]
> TypeOfContraction(X,1);
divisorial (K.C<0)
> TypeOfContraction(X,2);
flip
> WeightsOfFlip(X,2);
[
    [ 3, 2, -5, -1 ]
]
```

Example H118E18

We build a (nonsingular) variety X that is the projectivisation of the direct sum of line bundles $\mathcal{O}(0,0,0,1,1,1,1,2)$ on the projective line $\mathbf{P}^1$.

```
> X<[x]> := RationalScroll(Rationals(),1,[0,0,0,1,1,1,1,2]);
> X;
Toric variety of dimension 8
Variables: x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], x[9], x[10]
The components of the irrelevant ideal are:
    (x[10], x[9], x[8], x[7], x[6], x[5], x[4], x[3]), (x[2], x[1])
The 2 gradings are:
    1,  1,  0,  0,  0, -1, -1, -1, -1, -2,
    0,  0,  1,  1,  1,  1,  1,  1,  1,  1
```

We compute its nef cone simply to initiate all its Mori theoretic data.

```
> _ := NefCone(X);
```

We can consider various extreme rays of the Mori cone of X. (Our choice of parameters mean we consider rays $[C]$ for which $DC \leq 0$, where D is the zero divisor: that is, we consider all rays.)

```
> IsFlipping(X,1: divisor:=ZeroDivisor(X), inequality:="weak");
false
> IsFlipping(X,2: divisor:=ZeroDivisor(X), inequality:="weak");
true
```

One of the rays corresponds to the fibration of X onto $\mathbf{P}^1$. The other is a ray of (anti-)flipping type. We can make the antiflip.

```
> Y<[y]> := Flip(X,2: divisor:=ZeroDivisor(X), inequality:="weak");
> Y;
Toric variety of dimension 8
Variables: y[1], y[2], y[3], y[4], y[5], y[6], y[7], y[8], y[9], y[10]
The components of the irrelevant ideal are:
    (y[10], y[9], y[8], y[7], y[6]), (y[5], y[4], y[3], y[2], y[1])
The 2 gradings are:
    0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 2, 2, 2, 1, 1, 1, 1, 0
> IsNonsingular(Y);
false
> IsTerminal(Y);
true
```

This antiflip can be regarded as coming from a change in the linearisation of a geometric invariant theory quotient: from a linearisation like $(1,1)$ (between variables 2 and 3 in the given order) to one like $(2,3)$ (between variables 5 and 6). To understand the antiflip better, sometimes it helps to consider its weights (the relation between vertices on the star of the flipping locus, or, in geometric invariant theory terms, the weights of the local $\mathbf{G}_{\mathrm{m}}$ action that determine the flip).

```
> WeightsOfFlip(X,2: divisor:=ZeroDivisor(X), inequality:="weak");
[
    [ 1, 1, 0, 0, 0, -1, -1, -1, -1, -2 ],
    [ 0, 0, 1, 1, 1, 1, 1, 1, 1, 1 ]
]
```

Here we see that a $\mathbf{P}^1$ has been antiflipped in favour of a weighted $\mathbf{P}^4(1,1,1,1,2)$, which is the source of the singularity on Y.

```
MMP(X)
```

`type` MonStgElt *Default : "terminal"*

A sequence of toric varieties that are all the varieties visited by making any sequence of extremal contractions from the toric variety X (and, if necessary, making the corresponding flip). A second sequence records the maps, in each case first by a sequence of the indices of the domain and codomain, and second by a string that describes the map.

The parameter `type` indicates which extremal rays are considered. It can be `terminal`, `canonical` or `all`. In each case, only toric varieties having these singularities will be allowed as images of the maps. (In particular, if the default value `terminal` is chosen, then only true K_X-negative extremal contractions will be followed.)

Example H118E19

First make a toric variety X, in this case some blowup of $\mathbf{P}^3$.

```
> F := FanOfWPS([1,1,1,1]);
> G := Blowup(F, Ambient(F) ! [1,-1,1]);
> X := ToricVariety(Rationals(),G);
```

We compute all minimal model programs from X. There are two outputs: first a sequence containing those toric varieties encountered during these processes, and second a sequence containing all the maps encountered.

```
> models,mmp := MMP(X);
> #models;
3
> mmp;
[ [*
    [ 1, 2 ],
    divisorial (K.C<0)
*], [*
    [ 2, 3 ],
    map to point
*] ]
```

In this case there are three varieties. We could check that they are: (1) X itself, (2) $\mathbf{P}^3$, and (3) a point. We also see two maps: the first, labelled $[1,2]$, is the contraction of X back down to $\mathbf{P}^3$, and the second is the extremal contraction of $\mathbf{P}^3$ to a point.

The intrinsic call `MMP(X)` has a parameter, and the default is to search only for true absolute minimal model programs: those which proceed only by contracting extremal rays that are negative against the canonical class (and flipping them if necessary). To allow contractions of other extremal rays (and the possibility of requiring antiflips), we can set the parameter to `type:="all"`.

```
> models,mmp := MMP(X : type:="all");
> models;
[
    Toric variety of dimension 3
    Variables: $.1, $.2, $.3, $.4, $.5
    The components of the irrelevant ideal are:
        ($.5, $.4), ($.3, $.2, $.1)
    The 2 gradings are:
        1, 0, 0, 2, 1,
        1, 1, 1, 1, 0,
    Toric variety of dimension 3
    Variables: $.1, $.2, $.3, $.4
    The irrelevant ideal is:
        ($.4, $.3, $.2, $.1)
    The grading is:
        1, 1, 1, 1,
    Toric variety of dimension 3
    Variables: $.1, $.2, $.3, $.4, $.5
    The components of the irrelevant ideal are:
        ($.3, $.2), ($.5, $.4, $.1)
    The 2 gradings are:
        1, 0, 0, 2, 1,
        1, 1, 1, 1, 0,
    Toric variety of dimension 0,
    Toric variety of dimension 1
    Variables: $.1, $.2
    The irrelevant ideal is:
        ($.2, $.1)
    The grading is:
        1, 1
]
> mmp;
[ [*
    [ 1, 2 ],
    divisorial (K.C<0)
*], [*
    [ 1, 3 ],
    flop
*], [*
    [ 2, 4 ],
    map to point
*], [*
    [ 3, 1 ],
    flop
*], [*
    [ 3, 5 ],
    fibration (K.C<0)
*] ]
```

Now we also see a flop from X to a new toric variety Y (model number 3), and then Y admits Mori fibration to $\mathbf{P}^1$ (which we could check is model number 5).

118.8.3 Decomposition of Toric Morphisms

The title of this section is that of a well-known paper of Reid [Rei83]. It applies Mori theory to toric varieties relatively over a base to compute a relative minimal model and the relative canonical model of a toric variety. Rather than wrap this up in intrinsics, we show how to apply the various components of this package to realise Reid's result.

Example H118E20

We will compute a relative minimal model and a relative canonical model of the weighted projective space $\mathbf{P}^3(1,2,5,6)$, a 3-fold that does not have canonical (nor terminal) singularities.

```
> A := ProjectiveSpace(Rationals(),[1,2,5,6]);
> IsCanonical(A);
false
```

We find the relative models by running a minimal model program on a resolution of A relative to A itself (that is, we only allow morphisms $V \to W$ if they factor the given morphism $V \to A$). So we start by constructing a resolution.

```
> V0,f0 := Resolution(A);
> V0;
Toric variety of dimension 3
Variables: $.1, $.2, $.3, $.4, $.5, $.6, $.7, $.8, $.9, $.10, $.11, $.12
The irrelevant ideal is:
    [... omitted... ]
The 9 gradings are:
    0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0,
    0, 0, 1, 2, 0, 0, 0, 1, 0, 0, 0, 0,
    0, 0, 2, 3, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 1, 2, 3, 0, 0, 1, 0, 0, 0, 0, 0,
    0, 1, 3, 3, 0, 0, 0, 0, 0, 0, 0, 1,
    0, 1, 3, 4, 1, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 4, 5, 0, 1, 0, 0, 0, 0, 0, 0,
    1, 2, 5, 6, 0, 0, 0, 0, 0, 0, 0, 0
```

We see that the resolution made $8 = 9 - 1$ blowups. To compute everything, we would keep track of the morphisms (as we have started with f_0 above); for brevity, we forego that.

We build a function which detects which of the extremal contractions from a given V (starting with V_0) that factor $V \to A$. This code also reports whether the contraction is divisorial or of flipping type so that we can make the next step accordingly.

```
> raysOverA := func< W |
>     [ <i,TypeOfContraction(W,i)> :
>             i in [1..#ExtremalRays(W)] |
>     IsRegular(ToricVarietyMap(ExtremalRayContraction(W,i),A)) ] >;
```

And so we look for extremal rays over A that we can contract.

```
> raysOverA(V0);
[ <1, "divisorial (K.C<0)">, <2, "divisorial (K.C<0)">,
<3, "divisorial (K.C<0)"> ]
```

In other words, the extremal rays 1, 2 and 3 all determine extremal divisorial contractions. (If we were writing a faster routine, we would probably stop as soon as we'd found that ray 1 worked.) We can pick any of these: we choose ray 1 and repeat the process.

```
> V1 := ExtremalRayContraction(V0,1);
> raysOverA(V1);
[ <1, "divisorial (K.C<0)">, <2, "divisorial (K.C<0)">, <4, "divisorial
(K.C<0)"> ]
```

This time rays 1, 2, and 4 work; again these lead to divisorial contractions. We follow ray 1.

```
> V2 := ExtremalRayContraction(V1,1);
> raysOverA(V2);
[ <2, "divisorial (K.C<0)">, <3, "divisorial (K.C<0)"> ]
> V3 := ExtremalRayContraction(V2,2);
> raysOverA(V3);
[ <2, "divisorial (K.C<0)"> ]
> V4 := ExtremalRayContraction(V3,2);
> raysOverA(V4);
[ <2, "divisorial (K.C<0)"> ]
> V5<[w]> := ExtremalRayContraction(V4,2);
> assert #raysOverA(V5) eq 0;
```

There are no extremal rays over A to contract. We have reached a relatively minimal model, V_5; we check its singularities, although there is no need.

```
> V5;
Toric variety of dimension 3
Variables: w[1], w[2], w[3], w[4], w[5], w[6]
The components of the irrelevant ideal are:
    (w[6], w[5]), (w[6], w[3]), (w[5], w[4], w[2]), (w[3], w[1]), (w[4], w[2],
        w[1])
The 3 gradings are:
    0, 0, 1, 1, 0, 1,
    0, 1, 2, 3, 1, 0,
    1, 2, 5, 6, 0, 0
> IsTerminal(V5);
```

```
true
```

To continue on to a relatively canonical model we must consider rays that are trivial against the canonical class, not negative as we have so far. We need to modify the relative rays function; we simply weaken the inequality used when evaluating against the canonical class.

```
> weakraysOverA := func< W |
>     [ <i,TypeOfContraction(W,i : inequality:="weak")> :
>                  i in [1..#ExtremalRays(W:inequality:="weak")] |
>  IsRegular(ToricVarietyMap(ExtremalRayContraction(W,i:inequality:="weak"),A))]>;
```

And so we continue.

```
> weakraysOverA(V5);
[ <2, "divisorial (K.C=0)"> ]
> V6<[u]> := ExtremalRayContraction(V5,2 : inequality:="weak");
> assert #weakraysOverA(V6) eq 0;
```

After one canonically-trivial divisorial contraction there are no rays left to contract. We have reached the relative canonical model.

```
> V6;
Toric variety of dimension 3
Variables: u[1], u[2], u[3], u[4], u[5]
The components of the irrelevant ideal are:
    (u[5], u[3]), (u[4], u[2], u[1])
The 2 gradings are:
    0, 0, 1, 1, 1,
    1, 2, 5, 6, 0
> IsTerminal(V6);
false
> IsCanonical(V6);
true
```

Of course, in this toric setting, it would have been simpler to find the single short vector in the fan of A that was causing all the trouble and blow that up.

118.9 Schemes in Toric Varieties

The polynomials of the Cox ring of a toric variety X provide homogeneous coordinates on X that can be used to define subschemes of X. These subschemes are true Magma schemes, and so the usual scheme machinery works for them. However, there is a substantial caveat to this for the first version of the toric geometry package: affine patches have not been installed systematically, and so scheme machinery that uses affine patches of schemes will not work.

118.9.1 Construction of Subschemes

```
Scheme(X,f)
```

The subscheme of the toric variety X defined by the polynomial f from the Cox ring of X.

```
Scheme(X,Q)
```

The subscheme of the toric variety X defined by the sequence Q of polynomials from the Cox ring of X.

Example H118E21

Toric varieties are the natural ambient space for many varieties. Here we review the example of a trigonal curve from the Schemes chapter (it is self-contained here).
First make a curve. (This curve is in fact trigonal—it admits a 3-to-1 cover of the projective line. Once you've had that thought, it's actually pretty clear: the defining equation is a cubic in y. But there's more to it than just being trigonal, as we will see.)

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^8 + x^4*y^3*z + z^8);
> Genus(C);
8
```

This curve is of general type (that is, its genus is at least 2), so we can consider the canonical map: that will either be an embedding or a 2-to-1 map to a projective line.
We make the canonical map take its image in a toric variety.

```
> eqns := Sections(CanonicalLinearSystem(C));
> X<[a]> := ProjectiveSpace(Rationals(),7);
> f := map< P -> X | eqns >;
> V := f(C);
> V;
Curve over Rational Field defined by
a[1]^3 + a[2]^2*a[4] + a[1]*a[8]^2,
a[1]^2*a[3] + a[2]^2*a[6] + a[3]*a[8]^2,
a[1]^2*a[5] + a[2]*a[4]*a[6] + a[5]*a[8]^2,
a[1]*a[4]*a[6] - a[2]^2*a[7],
a[1]*a[6]^2 - a[2]^2*a[8],
a[2]*a[6]^2 + a[1]^2*a[7] + a[7]*a[8]^2,
a[4]*a[6]^2 + a[1]^2*a[8] + a[8]^3,
a[2]*a[3] - a[1]*a[4],
a[3]^2 - a[1]*a[5],
a[3]*a[4] - a[1]*a[6],
a[4]^2 - a[2]*a[6],
a[2]*a[5] - a[1]*a[6],
a[3]*a[5] - a[1]*a[7],
a[4]*a[5] - a[2]*a[7],
a[5]^2 - a[1]*a[8],
a[3]*a[6] - a[2]*a[7],
a[5]*a[6] - a[2]*a[8],
a[3]*a[7] - a[1]*a[8],
a[4]*a[7] - a[2]*a[8],
a[5]*a[7] - a[3]*a[8],
a[6]*a[7] - a[4]*a[8],
a[7]^2 - a[5]*a[8]
```

All those binomial equations suggest that V lies on a toric variety embedded in $X = \mathbf{P}^7$. We can recover this toric variety and its map to X.

```
> W,g := BinomialToricEmbedding(V);
> Y<[b]> := Domain(g);
> Y;
Toric variety of dimension 2
Variables: b[1], b[2], b[3], b[4]
The components of the irrelevant ideal are:
    (b[3], b[2]), (b[4], b[1])
The 2 gradings are:
    0, 1, 1, 0,
    1, 0, 2, 1
```

It is a well-known consequence of (geometric) Riemann–Roch that trigonal curves lie on scrolls in their canonical embeddings. Exactly which scroll is an intrinsic property of the particular curve: the Maroni invariant of a trigonal curve can be realised as the twist that occurs in the scroll, in this case 2 (visible in the last line of output above).

This makes good sense: the scroll Y has a natural map to $\mathbf{P}^1$, and the equation of the curve W is a cubic in the fibre variables $b[2], b[3]$ so defines a 3-to-1 cover of the base.

```
> I := Saturation(DefiningIdeal(W),IrrelevantIdeal(Y));
> Basis(I);
[
     b[1]^8*b[2]^3 + b[1]*b[3]^3*b[4] + b[2]^3*b[4]^8
]
```

The need for saturation is already visible in the equations of V: all those cubics are really multiples of a single cubic on the scroll by irrelevant ideals, but written in the coordinates of the projective space.

118.10 Bibliography

[**Cox95**] David A. Cox. The homogeneous coordinate ring of a toric variety. *J. Algebraic Geom.*, 4(1):17–50, 1995.

[**Dan78**] V. I. Danilov. The geometry of toric varieties. *Uspekhi Mat. Nauk*, 33(2(200)): 85–134, 247, 1978.

[**Ful93**] William Fulton. *Introduction to toric varieties*, volume 131 of *Annals of Mathematics Studies.* Princeton University Press, Princeton, NJ, 1993. The William H. Roever Lectures in Geometry.

[**Oda88**] Tadao Oda. *Convex bodies and algebraic geometry*, volume 15 of *Ergebnisse der Mathematik und ihrer Grenzgebiete (3) [Results in Mathematics and Related Areas (3)].* Springer-Verlag, Berlin, 1988. An introduction to the theory of toric varieties, Translated from the Japanese.

[**Rei83**] Miles Reid. Decomposition of toric morphisms. In *Arithmetic and geometry, Vol. II*, volume 36 of *Progr. Math.*, pages 395–418. Birkhäuser Boston, Boston, MA, 1983.