

Handbook of Magma Functions

Part XV: Algebraic Geometry

Version 2.30-1

Sydney

October 15, 2026

Editors

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

Handbook Contributors

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

Production Editors

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

HTML Production

Claus Fieker *Allan Steel*

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

Contents

122 Schemes **1**

122.1 Introduction and First Examples . . . 1
122.2 Ambients . . . 9
122.3 Constructing Schemes . . . 16
122.4 Different Types of Scheme . . . 20
122.5 Basic Attributes of Schemes . . . 21
122.6 Function Fields and their Elements . . . 24
122.7 Rational Points and Point Sets . . . 26
122.8 Zero-dimensional Schemes . . . 29
122.9 Local Geometry of Schemes . . . 31
122.10 Classification and Normal Forms of Singularities . . . 34
122.11 Global Geometry of Schemes . . . 41
122.12 Base Change for Schemes . . . 44
122.13 Affine Patches and Projective Closure . . . 46
122.14 Arithmetic Properties of Schemes and Points . . . 50
122.15 Maps between Schemes . . . 55
122.16 Tangent and Secant Varieties and Isomorphic Projections . . . 81
122.17 Linear Systems . . . 85
122.18 Divisors . . . 99
122.19 Isolated Points on Schemes . . . 107
122.20 Advanced Examples . . . 113
122.21 Bibliography . . . 117

123 Coherent Sheaves **118**

123.1 Introduction . . . 118
123.2 Creation Functions . . . 119
123.3 Accessor Functions . . . 121
123.4 Basic Constructions . . . 122
123.5 Sheaf Homomorphisms . . . 124
123.6 Divisor Maps and Riemann-Roch Spaces . . . 125
123.7 Predicates . . . 128
123.8 Miscellaneous . . . 130
123.9 Examples . . . 130
123.10 Bibliography . . . 140

124 Algebraic Curves **141**

124.1 First Examples . . . 141
124.2 Ambient Spaces . . . 147
124.3 Algebraic Curves . . . 148
124.4 Local Geometry . . . 157
124.5 Global Geometry . . . 165
124.6 Maps and Curves . . . 169
124.7 Automorphism Groups of Curves . . . 172
124.8 Function Fields . . . 181
124.9 Divisors . . . 191
124.10 Linear Equivalence of Divisors . . . 201
124.11 Advanced Examples . . . 207

124.12 Curves over Global Fields . . . 210
124.13 Minimal Degree Functions and Plane Models . . . 214
124.14 Bibliography . . . 221

125 Resolution Graphs and Splice Diagrams 222
125.1 Introduction . . . 222
125.2 Resolution Graphs . . . 222
125.3 Splice Diagrams . . . 229
125.4 Translation Between Graphs . . . 231
125.5 Bibliography . . . 231

126 Algebraic Surfaces 232
126.1 Introduction . . . 232
126.2 Generalities . . . 232
126.3 General Surfaces . . . 236
126.4 Desingularisation by Blow Up . . . 253
126.5 Surfaces in $\mathbb{P}^3$ . . . 283
126.6 Del Pezzo Surfaces . . . 301
126.7 Degree 2 K3 Surfaces . . . 316
126.8 Bibliography . . . 327

127 Riemann Surfaces 329
127.1 Introduction to Riemann Surfaces . . . 329
127.2 Creation Functions . . . 329
127.3 Properties of Riemann Surfaces . . . 334
127.4 Points on Riemann Surfaces . . . 339
127.5 Divisors on Riemann Surfaces . . . 343
127.6 Abel–Jacobi Map . . . 344
127.7 Period Matrix Functions . . . 347
127.8 Bibliography . . . 348

128 Hilbert Series of Polarised Varieties 349
128.1 Introduction . . . 349
128.2 Hilbert Series and Graded Rings . . . 351
128.3 Baskets of Singularities . . . 355
128.4 Generic Polarised Varieties . . . 360
128.5 Subcanonical Curves . . . 362
128.6 K3 Surfaces . . . 363
128.7 Weil Polynomials . . . 364
128.8 Point Counting on Degree Two K3 Surfaces . . . 366
128.9 The K3 Database . . . 367
128.10 Fano 3-folds . . . 371
128.11 Calabi–Yau 3-folds . . . 372
128.12 Building Databases . . . 372
128.13 Bibliography . . . 374

129 Toric Varieties 376
129.1 Introduction and First Examples . . . 376
129.2 Fans in Toric Lattices . . . 382
129.3 Geometrical Properties of Cones and Polyhedra . . . 390
129.4 Toric Varieties . . . 392
129.5 Cox Rings . . . 395
129.6 Invariant Divisors and Riemann-Roch Spaces . . . 399
129.7 Maps of Toric Varieties . . . 405
129.8 The Geometry of Toric Varieties . . . 407
129.9 Schemes in Toric Varieties . . . 413
129.10 Bibliography . . . 415

References 416

Index **420**

Chapter 122

Schemes

122.1 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 *Algebraic Curves* 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.

122.1.1 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: Creating an Affine Space (ex-9637c4)

```
> 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.

122.1.2 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.

Example: Defining the Twisted Cubic `(ex-1db4b0)`

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.

```
> 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 *Constructing Schemes* for more details).

122.1.3 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.)

Example: Creating Points by Coercion (ex-c1415a)

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.

```
> 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.

122.1.4 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: Affine Patches of the Projective Plane (ex-e6283e)

```
> 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
```

(continues on next page)

(continued from previous page)

```
> 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.

122.1.5 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: Map from the Line to the Plane (ex-8febb0)

```
> 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 := PointsOverAlgebraicClosure(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 *Algebraic Curves*.

122.1.6 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: Linear System of Plane Conics (ex-7c74bf)

```
> 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);
```

(continues on next page)

(continued from previous page)

```
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
```

122.1.7 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 *Algebraic Curves*. 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.

122.2 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.

122.2.1 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): Rng, RngIntElt -> Aff`

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): Rng, RngIntElt -> Prj`

`ProjectiveSpace(k, W): Rng, [RngIntElt] -> Prj`

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.

`ProjectiveLine(k): RngIntRes -> SetIndx, UserProgram`

ProjectiveLine(k): RngOrdRes -> SetIndx, UserProgram

Type: MonStgElt Default: "Vector"

Given a quotient ring R of an order O of some number field, returns an indexed set containing standard representatives $(u, v) \in \mathbf{P}^1(R)$, where $u, v \in O$. Also returns a function which identifies the standard representative of any given element $(u', v') \in \mathbf{P}^1(R)$. The optional argument Type should be either "Vector" or "Matrix", which determines the type of the elements in the returned set.

ProjectiveLineProcess(V): ModTupFld[FldFin] -> ProcPL

ProjectiveLineProcess(k, n): FldFin, RngIntElt -> ProcPL

Given a vector space V over a finite field k, or its dimension n, returns a process which enumerates the lines in V (elements of $\mathbf{P}(V)$).

P: ProcPL -> RngIntElt

The number of lines in $P = \mathbf{P}(V)$.

Next(P): ProcPL -> ModTupFldElt

A vector spanning the next line in the process $P = \mathbf{P}(V)$. Returns the zero vector if no more elements are left.

AffineSpace(R): RngMPol -> Aff

Spec(R): RngMPol -> Aff

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): RngMPol -> Prj

Proj(R): RngMPol -> Prj

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): Sch, [MonStgElt]

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: Sch, RngIntElt -> RngMPolElt

Name(A, i): Sch, RngIntElt -> RngMPolElt

The ith coordinate function of A as an element of the coordinate ring of A.

Example: Naming Affine Coordinates (ex-342021)

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
```

(continues on next page)

(continued from previous page)

```
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: Sch, Sch -> BoolElt`

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.

122.2.2 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

$$u^5x, u^4vx, \ldots, v^5x, uy, vy.$$

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 $\mathbb{F}_n$. 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 $\mathbb{F}_n$ — 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

$$\begin{pmatrix} 1 & \ldots & 1 & * & \ldots & * & \ldots & & & & \\ 0 & \ldots & 0 & 1 & \ldots & 1 & * & \ldots & & & \\ \vdots & & & & & & & & & & \\ 0 & \ldots & 0 & 0 & \ldots & 0 & 0 & \ldots & 0 & 1 & \ldots & 1 \end{pmatrix}$$

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): Sch, Sch -> Sch, SeqEnum`

The product of the ambient spaces A and B together with a sequence containing the two projection maps.

`RuledSurface(k, a, b): Rng, RngIntElt, RngIntElt -> PrjScrl`

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): Rng, RngIntElt -> PrjScrl`

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): Rng, SeqEnum -> PrjScrl`

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): Rng, SeqEnum -> PrjScrl`

If $N = [n_1, \ldots, n_r]$ is a sequence of positive integers this returns the product of ordinary projective spaces

$$\mathbf{P}^{n_1} \times \cdots \times \mathbf{P}^{n_r}$$

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.

`SegreProduct(Xs): SeqEnum[Sch] -> Sch, SeqEnum`

Computes the product X of a finite sequence of schemes Xs lying in ordinary projective space. The scheme X is constructed in *ordinary* projective space. It is embedded there via an iterated Segre embedding (see Ex. 2.14, Section 2, Chapter 1 of [Har77]). This intrinsic is provided because it is often easier to work in ordinary projective space. However, the user should be warned that the dimension of the ambient increases markedly with Segre embeddings. If the r schemes $Xs[i]$ lie in $\mathbf{P}^{n_i}$, then X will lie in $\mathbf{P}^N$ where N is $(n_1+1)*(n_2+1)*\ldots(n_r+1)-1$.

A sequence containing the r projection maps from X to the $Xs[i]$ is also returned.

`SegreEmbedding(X): Sch -> Sch, MapIsoSch`

The scheme X should lie in a product projective ambient (or be the ambient itself!). Computes and returns the image Y of X in ordinary projective space under the iterated Segre embedding (see the preceding intrinsic) along with a scheme isomorphism from X to Y. This is a specialised intrinsic that is generally much faster than using the general machinery to construct the Segre map on the ambient of X and ask for the image of X.

Example: Segre Embedding of a Product (ex-b233a4)

In the following example, we Segre embed the product of an elliptic curve E with itself into ordinary projective space using the two intrinsics.

```
> Q := RationalField();
> P2<x,y,z> := ProjectiveSpace(Q,2);
> E := Curve(P2,y^2*z-x^3-x^2*z-z^3);
> ExE := SegreProduct([E,E]);
> P8<a,b,c,d,e,f,g,h,i> := Ambient(ExE);
> ExE;
```

(continues on next page)

(continued from previous page)

```
Scheme over Rational Field defined by
-f*h + e*i,
-c*h + b*i,
-f*g + d*i,
-e*g + d*h,
-c*g + a*i,
-b*g + a*h,
-c*e + b*f,
-c*d + a*f,
-b*d + a*e,
-g^3 - g^2*i + h^2*i - i^3,
-d^3 - d^2*f + e^2*f - f^3,
-c^3 - c^2*i + f^2*i - i^3,
-b^3 - b^2*h + e^2*h - h^3,
-a^3 - a^2*g + d^2*g - g^3,
-a^3 - a^2*c + b^2*c - c^3
> // or we could have started with ExE in product projective space
> P22<x,y,z,s,t,u> := ProductProjectiveSpace(Q,[2,2]);
> EE := Scheme(P22,[y^2*z-x^3-x^2*z-z^3, t^2*u-s^3-s^2*u-u^3]);
> EE;
Scheme over Rational Field defined by
-x^3 - x^2*z + y^2*z - z^3,
-s^3 - s^2*u + t^2*u - u^3
> ExE_1 := SegreEmbedding(EE);
> // transfer ExE_1 to the Ambient of ExE to compare
> ExE_1 := Scheme(P8,[Evaluate(pol,[a,b,c,d,e,f,g,h,i]) : pol in
>             DefiningPolynomials(ExE_1)]);
> ExE eq ExE_1;
true
```

122.2.3 Functions and Homogeneity on Ambient Spaces

CoordinateRing(A): Sch -> Rng

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): Sch -> FldFunFracSch

The function field of the ambient space A of dimension 1. This is a field isomorphic to the field of fractions of the coordinate ring of A. While it is possible to give input of dimension greater than 1 for backward compatibility, it is preferrable to instead use `FieldOfFractions(CoordinateRing(X))`.

HasFunctionField(A): Sch -> BoolElt

Gradings(X): Sch -> SeqEnum

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): Sch -> RngIntElt

NGrad(X): Sch -> RngIntElt

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): Sch -> RngIntElt

Length(X): Sch -> RngIntElt

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): Sch -> [RngIntElt]

The lengths of the groups of ones in the gradings of a scroll X.

IsHomogeneous(X, f): Sch, RngMPolElt -> BoolElt

IsHomogeneous(X, f): Sch, FldFunRatMElt -> BoolElt

Returns **true** if and only if the polynomial or rational function f is homogeneous with respect to all of the gradings on the scheme X. The rational function must be of the coordinate ring of the ambient of X and **true** is returned if and only if f is a quotient of polynomials homogeneous with respect to all gradings.

Multidegree(X, f): Sch, RngMPolElt -> SeqEnum

Degrees(X, f): Sch, RngMPolElt -> SeqEnum

The sequence of homogeneous degrees of the polynomial f with respect to the gradings on the scheme X.

122.2.4 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 do not 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 *Rational Points and Point Sets*, should be taken as the definitive reference.

A ! [a,b,...]: Sch, [RngElt] -> Pt

A(L) ! [a,b,...]: SetPt, [RngElt] -> Pt

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: Equality of Points (ex-68807d)

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;
```

(continues on next page)

(continued from previous page)

```
>> 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): Aff -> Pt`

The origin of the affine space A.

`Simplex(A): Prj -> SeqEnum`

The sequence of points of the ambient space A having coordinates $(1, 0, \ldots, 0)$, ...$(0, \ldots, 0, 1)$, and $(1, \ldots, 1)$ whether A is affine or projective space.

`Coordinates(p): Pt -> SeqEnum`

The sequence of ring elements corresponding to the coordinates of the point p.

`p[i]: Pt, RngIntElt -> RngElt`

`Coordinate(p, i): Pt, RngIntElt -> SeqEnum`

The ith coordinate of the point p.

`p @ f: Pt, FldFunFracSchElt -> RngElt`

`f(p): Pt, FldFunFracSchElt -> RngElt`

`Evaluate(f, p): RngElt, Pt -> RngElt`

Evaluate the function f of the function field of the scheme X or its ambient at point p which lies on X.

Example: Evaluating Functions at Points (ex-56a4f5)

Although there are many rings which may appear as function fields in various contexts, their elements can all be evaluated at points. (Section *Function Fields* 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> := FieldOfFractions(CoordinateRing(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
```

122.3 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 $(x_1, \ldots, x_n)$. 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): Sch, RngMPolElt -> Sch`

`Scheme(X, F): Sch, SeqEnum -> Sch`

`Scheme(X, I): Sch, RngMPol -> Sch`

`Scheme(X, Q): Sch, RngMPolRes -> Sch`

```
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): Sch, RngMPolElt -> Clstr

Cluster(X, F): Sch, SeqEnum -> Clstr

Cluster(X, I): Sch, RngMPol -> Clstr

Cluster(X, Q): Sch, RngMPolRes -> Clstr

```
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: Creating Schemes (ex-dc463d)

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): RngMPolRes -> Sch, Aff

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): RngMPolRes -> Sch, Prj

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): Sch -> Sch

EmptySubscheme(X): Sch -> Sch, MapSch

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: Sch, Sch -> Sch

Intersection(X, Y): Sch, Sch -> Sch

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: Sch, Sch -> Sch

Union(X, Y): Sch, Sch -> Sch

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: [Sch] -> Sch`

The union of the schemes in the sequence S in their common ambient space.

`Difference(X, Y): Sch, Sch -> Sch`

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.

`Complement(X, Y): Sch, Sch -> Sch`

This is like `Difference` except that *all* points of (`X meet Y`) are removed from scheme X, irrespective of multiplicity, and the closure is returned. It corresponds to saturation rather than colon ideal.If X is saturated then the result is as well and is marked as such.

`RemoveLinearRelations(X): Sch -> Sch, MapIsoSch`

Convenience function that takes linear relations between variables on X and uses them to eliminate variables. The intrinsic is currently only available 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.

`Blowup(X, Y): Sch, Sch -> Sch, MapSch`

`Blowup(X, p): Sch, Pt -> Sch, MapSch`

```
Ordinary: BoolElt                    Default: true
Xirred  : BoolElt                    Default: true
```

The first intrinsic constructs the scheme obtained from blowing up subscheme Y of scheme X (see Section 7, Chapter II of [Har77]). The second is a user convenience special case that blows up the subscheme consisting of a point p (and all of its conjugates if it is not defined over the base field) in a pointset $X(K)$.

Currently X must lie in an ambient that is affine, ordinary projective, or product projective. If parameter `Ordinary` is `true` (the default), and X is projective, then the result of the blow-up is embedded in ordinary projective space via the Segre embedding. Otherwise, the result will lie in an ambient that is the direct product of the ambient of X and an ordinary projective space.

The final saturation of the blowup procedure can be performed more efficiently when the scheme X is known to be irreducible. Since this is the commonest case for users, it is assumed by default. If it is not known, the parameter `Xirred` should be set to `false`. Note that if X is not irreducible, the intrinsic can actually fail to perform the computation in which case an appropriate error message is given. Technically, for this implementation, the success relies on finding a relative set of generators for the ideal of Y modulo the ideal of X which give non zero-divisors on X.

The second return value in each case is the blow-down map from the blow-up back to X.

An example is given below. For another example, see the second minimal model example following the description of *CanonicalCoordinateIdeal* in the chapter on algebraic surfaces.

The implementation makes use of the `ReesIdeal(P, I): RngMPol, RngMPol → RngMPol, Map` intrinsic.

`LocalBlowUp(X, Y): Sch, Sch -> SeqEnum`

A useful variant of the above blow up intrinsics is this local version that does not globally embed the result but returns a sequence of affine patches that cover the blow-up. Here X must be affine and Y, as usual is the subscheme to be blown up. If X is not a variety, it is assumed that the defining polynomials of Y that do not lie in the defining ideal of X, are non-zero divisors modulo that ideal. The return value is a sequence of pairs $< S, mp >$ where S is an affine scheme which is an open patch of (i.e. it is Zariski-open in) the complete blow-up and mp is the blow down map from S back to X. Here, the birational blow-down maps come with their inverses.

Example: Difference of Schemes (ex-51f816)

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.

Example: Blowing Up a Line in Projective Space (ex-0b76be)

We give an example of blowing-up a line (isomorphic to $\mathbf{P^1}$) in projective 3-space $\mathbf{P^3}$. The inverse image of the line in the blow-up X is isomorphic to the ruled surface $\mathbf{P^1} \times \mathbf{P^1}$ which is embedded in X as a quadric cone.

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> L := Scheme(P3,[z,t]); //the line to be blown up
> X,mp := Blowup(P3,L);
> P<[a]> := Ambient(X); X;
Scheme over Rational Field defined by
a[5] - a[8],
-a[6]*a[7] + a[5]*a[8],
-a[4]*a[7] + a[3]*a[8],
-a[2]*a[7] + a[1]*a[8],
-a[4]*a[5] + a[3]*a[6],
-a[2]*a[5] + a[1]*a[6],
-a[2]*a[3] + a[1]*a[4]
> Y := L @@ mp;  // the inverse image of L
> Dimension(Y);
2
> MinimalBasis(Ideal(Y));
[
    a[8],
    a[7],
    a[6],
    a[5],
```

(continues on next page)

(continued from previous page)

```
    a[2]*a[3] - a[1]*a[4]
]
```

Saturate(~X): Sch

If the scheme X is projective and is not already saturated, saturate its defining ideal.

AssignNames(~X, N): Sch, SeqEnum

Assign the strings in the sequence N to the ambient coordinate functions of the scheme X.

X . i: Sch, RngIntElt -> RngMPolElt

Name(X, i): Sch, RngIntElt -> RngMPolElt

The ith coordinate function of the ambient space of the scheme X. The dot notation `X.i` may also be used.

122.4 Different Types of Scheme

As discussed briefly in Section *Aside: Types of Schemes*, 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): Sch -> BoolElt

Returns `true` if and only if the scheme X is an affine space.

IsProjective(X): Sch -> BoolElt

Returns `true` if and only if the scheme X is a projective space. Projective space here includes the case of scrolls.

IsOrdinaryProjectiveSpace(X): Sch -> BoolElt

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): Sch -> BoolElt

Return `true` if the scheme X is an ambient space.

IsCluster(X): Sch -> BoolElt, Clstr

Returns `true` if and only if the scheme X is a zero-dimensional scheme (but not the empty scheme). See Section *Zero-dimensional Schemes* for intrinsics which apply to clusters.

IsCurve(X): Sch -> BoolElt, Crv

Returns `true` if and only if X is a one-dimensional scheme. See Chapter *Algebraic Curves* for intrinsics which apply to curves.

IsPlaneCurve(X): Sch -> BoolElt, CrvPln

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): Sch -> BoolElt, CrvCon

Returns `true` if and only if the scheme X is a curve (in the sense of `IsCurve(X)`) which is non-singular and defined by an equation of degree 2. See Chapter Rational Curves and Conics for intrinsics which apply to such conics.

IsRationalCurve(X): Sch -> BoolElt, CrvRat

Returns `true` if and only if the scheme X is a curve (in the sense of `IsCurve(X)`) which has genus 0. See Chapter Rational Curves and Conics for intrinsics which apply to rational curves.

`IsHyperellipticCurve(X): Sch -> BoolElt, CrvHyp`

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 *Algebraic Curves* 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 Hyperelliptic Curves for more information on hyperelliptic curves.

`IsModularCurve(X): Sch -> BoolElt`

Return `true` if and only if the scheme X is a curve of type `CrvMod`. See Chapter Modular Curves for more information on modular curves.

122.5 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.

122.5.1 Functions of the Ambient Space

`AmbientSpace(X): Sch -> Sch`

`Ambient(X): Sch -> Sch`

The ambient space containing the scheme X.

`IdenticalAmbientSpace(X, Y): Sch, Sch -> BoolElt`

Returns whether schemes X and Y lie in the same ambient space.

`SuperScheme(X): Sch -> Sch`

The scheme X was created as a subscheme of.

`BaseRing(X): Sch -> Rng`

`CoefficientRing(X): Sch -> Fld`

The base ring of the scheme X.

`HasGCD(X): Sch -> BoolElt`

Returns whether GCDs can be computed for multivariate polynomials over the base ring of scheme X.

`HasGroebnerBasis(X): Sch -> BoolElt`

Returns whether Groebner bases can be computed for multivariate polynomial ideals over the base ring of scheme X.

`HasResultant(X): Sch -> BoolElt`

Returns whether resultants can be computed for multivariate polynomials over the base ring of scheme X.

`BaseField(X): Sch -> Fld`

`CoefficientField(X): Sch -> Fld`

The base ring of the scheme X if it is a field, otherwise an error.

`IsAffine(X): Sch -> BoolElt`

Returns `true` if and only if the ambient space of the scheme X is affine.

`IsProjective(X): Sch -> BoolElt`

Returns `true` if and only if the ambient space of the scheme X is projective.

`IsOrdinaryProjective(X): Sch -> BoolElt`

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): Sch -> BoolElt`

Return `true` if the ambient of the scheme X is 2-dimensional.

`IsSaturated(X): Sch -> BoolElt`

Returns `true` if and only if the current defining ideal of the scheme X, as returned by `DefiningIdeal(X)` is saturated (see Section *Constructing Schemes*).

122.5.2 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 does not involve ideal theory overheads and certainly will not call any Gröbner basis functions.

`DefiningPolynomials(X): Sch -> SeqEnum`

`Polynomials(X): Sch -> SeqEnum`

`DefiningEquations(X): Sch -> SeqEnum`

`Equations(X): Sch -> SeqEnum`

The defining polynomials for the ideal of the scheme X.

`DefiningPolynomial(X): Sch -> RngMPolElt`

`Polynomial(X): Sch -> RngMPolElt`

`DefiningEquation(X): Sch -> RngMPolElt`

`Equation(X): Sch -> RngMPolElt`

The defining polynomial of the scheme X if it is a hypersurface. If X is not a hypersurface, an error is reported.

`DefiningIdeal(X): Sch -> RngMPol`

`Ideal(X): Sch -> RngMPol`

The ideal of a multivariate polynomial ring defining the scheme X.

`CoordinateRing(X): Sch -> RngMPol`

The quotient of the coordinate ring of the ambient space of the scheme X by the ideal of X.

`Curve(X): Sch -> Crv`

The smallest scheme in the inclusion chain above the scheme X which is a curve.

`GroebnerBasis(X): Sch -> SeqEnum`

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): Sch -> [ RngMPolElt ]`

Return a minimal basis of the defining ideal of the scheme X, that is, a sequence of polynomials for which no proper subsequence 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): Sch -> BoolElt, RngMPolElt`

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 polynomials if possible. The polynomial is returned as a second value.

`NoCommonComponent(X, Y): Sch, Sch -> BoolElt`

Returns whether hypersurfaces X and Y lying in the same ambient space have a common irreducible component.

`CommonComponent(X, Y): Sch, Sch -> Sch`

For hypersurfaces X and Y lying in the same ambient space, returns the (possibly empty) maximal hypersurface lying in the intersection. This is simply computed as the hypersurface defined by the GCD of the defining polynomials of X and Y.

`JacobianIdeal(X): Sch -> RngMPol`

The ideal of partial derivatives of the polynomials which define the scheme X. See comment below about `MinimalBasis`.

`JacobianMatrix(X): Sch -> ModMatRngElt`

The matrix $(\partial f_i/\partial x_j)$ of partial derivatives of the defining polynomials of the scheme X. Note that this can use the `MinimalBasis` instead of the `DefiningEquations` if the former has fewer elements. (One can directly call `JacobianMatrix` on the sequence of polynomials given by `DefiningEquations`, if that is desired).

`HessianMatrix(X): Sch -> ModMatRngElt`

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: Sch, Sch -> BoolElt`

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): Sch, Sch -> BoolElt`

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): Sch -> BoolElt`

Return `true` if the scheme X is defined by linear equations, possibly after taking a Gröbner basis.

Example: Testing Scheme Inclusion and Equality `(ex-571415)`

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
```

122.6 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 of dimension 1 (and therefore of affine spaces of dimension 1) 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 *Maps between Schemes*) 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): FldFunFracSch -> Sch`

Return the (projective) scheme F is the function field of.

`IntegerRing(F): RngFrac -> Rng`

`Integers(F): RngFrac -> Rng`

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): RngFrac, [MonStgElt]`

Assign the strings in S to be the names of the integer ring of F.

`F ! g: FldFunFracSch, RngElt -> FldFunFracSchElt`

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: FldFunFracSch, RngIntElt -> FldFunFracSchElt`

Return the ith indeterminate of the coordinate ring of the scheme of F as an element of the function field F.

`ProjectiveFunction(f): FldFunFracSchElt -> FldFracElt`

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): FldFunFracSchElt -> FldFunRatMElt`

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): FldFunFracSchElt, Sch -> FldFracElt`

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): RngFracElt -> RngElt`

`Denominator(f): RngFracElt -> RngElt`

Given an element f of a function field of a scheme, return the numerator or denominator of f.

`f * g: RngFracElt, RngFracElt -> RngFracElt`

`f + g: RngFracElt, RngFracElt -> RngFracElt`

`f - g: RngFracElt, RngFracElt -> RngFracElt`

`- f: RngFracElt -> RngFracElt`

`f / g: RngFracElt, RngFracElt -> RngFracElt`

`f ^ n: RngFracElt, RngIntElt -> RngFracElt`

`f eq g: RngFracElt, RngFracElt -> BoolElt`

`IsZero(f): RngFracElt -> BoolElt`

`IsOne(f): RngFracElt -> BoolElt`

`IsMinusOne(f): RngFracElt -> BoolElt`

`IsUnit(f): RngFracElt -> BoolElt`

`IntegralSplit(f, X): FldFunFracSchElt, Sch -> RngMPolElt, RngMPolElt`

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): FldFunFracSchElt, Sch -> MPolElt`

The first return value of `IntegralSplit(f, X)`.

`Denominator(f, X): FldFunFracSchElt, Sch -> MPolElt`

The second return value of `IntegralSplit(f, X)`.

Example: Converting Function Field Elements (ex-5c8a0c)

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): FldFunFracSchElt, Sch -> FldFunFracSchElt`

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): Sch -> Pt`

Returns a point in the pointset `X(FunctionField(X))` of the scheme X, whose coordinates generate `FunctionField(X)`.

122.7 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 *Zero-dimensional Schemes* 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 *Prelude to Points* for descriptions of these and some other basic functions.

`X(L): Sch, Rng -> SetPt`

`PointSet(X, L): Sch, Rng -> SetPt`

`X(m): Sch, Map -> SetPt`

`PointSet(X, m): Sch, Map -> SetPt`

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: SetPt, SetPt -> BoolElt`

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): SetPt -> Sch`

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): SetPt -> Crv`

The smallest scheme in the inclusion chain above the scheme associated to the point set P which is a curve.

`Ring(P): SetPt -> Rng`

The ring L associated to the point set P where P is of the form $X(L)$ for some scheme X.

`RingMap(P): SetPt -> Map`

The map from the base ring of the scheme of P to the ring of the pointset P.

`X ! Q: Sch, SeqEnum -> Pt`

`X(L) ! Q: SetPt, SeqEnum -> Pt`

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: Pt, Pt -> BoolElt`

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: Pt, Sch -> BoolElt`

Returns `true` if and only if the point p lies in the scheme X or is coercible into it.

`Scheme(p): Pt -> Sch`

The scheme on which the point p lies.

`Curve(p): Pt -> Crv`

The smallest scheme in the inclusion chain above the scheme on which the point p lies which is a curve.

`Q in X: SeqEnum, Sch -> BoolElt`

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: Setq, Sch -> BoolElt`

Returns `true` if and only if all points of the set S lie in the scheme X or are coercible into it.

`RationalPoints(X): Sch -> SetIndx`

`RationalPoints(X, L): Sch, Rng -> SetIndx`

`Points(X): Sch -> SetIndx`

`Points(X, L): Sch, Rng -> SetIndx`

```
Bound               : RngIntElt                    Default: 1000
SkipSingularityCheck: BoolElt                      Default: false
```

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()`.

If the parameter SkipSingularityCheck is set to true, then the initial check for singularities is skipped at the beginning so that the function does not fail on singular inputs.

RationalPointsByFibration(X): Sch -> SetIndx

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): SetPt -> Pt

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): Sch -> BoolElt, Pt

HasNonsingularPoint(X, L): Sch, FldFin -> BoolElt, Pt

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.

PointsModPN(S, P, N): Sch, ., RngIntElt -> SetIndx

The points of the affine scheme S modulo P^N, returned as an indexed set of points of S base changed to the residue ring O/P^N, where O is the base ring of S. Here P is a principal prime ideal of O, given either as such an ideal or, when O is the ring of integers, as a prime integer, and N is a positive integer.

The points of S over the residue field of P are found first and then Hensel lifted through the powers of P; accordingly S must be affine and P must be principal. RationalPoints calls PointsModPN automatically when its argument is a scheme defined over a residue ring.

PointsModPN(S, M): Sch, RngIntElt -> SetIndx

PointsModPN(S, M): Sch, RngOrdIdl -> SetIndx

The points of the affine scheme S over the residue ring O/M, where O is the base ring of S and M is the modulus. Two overloads cover the two cases: O is the ring of integers and M is a positive integer, or O is a number-field order and M is a nonzero ideal of O. The modulus M is factored into prime powers and the points over each O/P^e are computed by the three-argument form; the results are recombined via the Chinese Remainder Theorem. Accordingly S must be affine and each prime divisor of M must be principal.

Example: Points on a Scheme over a Finite Field (ex-4f3a7a)

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` 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.

122.8 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): Pt -> Clstr`

`Cluster(X, p): Sch, Pt -> Clstr;`

`Cluster(S): SetEnum -> Clstr`

`Cluster(S): SetIndx -> Clstr`

`Cluster(X, S): Sch, SetEnum -> Clstr`

`Cluster(X, S): Sch, SetIndx -> Clstr`

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): Sch -> SetEnum`

`RationalPoints(Z, L): Sch, Rng -> SetEnum`

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.

`PointsOverAlgebraicClosure(Z): Clstr -> SetIndx, Fld`

The points of the cluster Z over an algebraic closure L of its base field, together with L. The base field must be the rational field, a finite field, or an algebraically closed field. If the base field is algebraically closed or Z is empty, L is the base field itself. `PointsOverSplittingField` returns the same points over a number field when the base field is the rational field, and over an explicit finite extension when it is a finite field.

`PointsOverSplittingField(Z): Clstr -> SetIndx, Fld`

`Absolute: BoolElt`	`Default: false`

The points of the cluster Z over a splitting field L, a finite extension of the base field, together with L. The base field must be the rational field, a finite field, or an algebraically closed field. Over an algebraically closed field this intrinsic behaves as `PointsOverAlgebraicClosure`. If Z is empty, L is the base field.

Over the rational field, L is a number field, or the rational field itself when no extension is needed. The points are found in an algebraic closure, as by `PointsOverAlgebraicClosure`, and their coordinates are then mapped into L. By default the number field is a tower with one extension for each variable of that algebraic closure. If the parameter `Absolute` is set to `true`, it is an absolute number field instead, which is usually more expensive both to construct and to compute with.

Over a finite field, L is a finite extension of the base field, possibly of degree 1. Over either field, L need not have minimal degree, and constructing it and computing with it may be expensive.

Example: Points of a Cluster over a Number Field `(ex-e7b0a1)`

We compute the points of a zero-dimensional scheme over the rationals. The splitting field is returned as a number field.

```
> P5<x, y, z, w, u, t> := ProjectiveSpace(Rationals(), 5);
> eqns := [
> w^2 - 37852414/35772413*u^2 - 22528936/35772413*w*t
>   + 5679939/35772413*u*t - 7111409/35772413*t^2,
> w*u + 52686636/35772413*u^2 + 62311891/71544826*w*t
>   - 17906287/35772413*u*t + 30283007/71544826*t^2,
> x + 163/1691*w - 668/1691*u + 210/1691*t,
> y + 696/1691*w + 1287/1691*u + 46/1691*t,
> z - 372/1691*w + 41/1691*u - 666/1691*t];
> Z := Scheme(P5, eqns);
> pts, L := PointsOverSplittingField(Z);
> Type(L);
FldNum
> #pts;
4
```

`HasPointsOverExtension(X): Sch -> BoolElt`

`HasPointsOverExtension(X, L): Sch, Rng -> BoolElt`

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): Clstr -> RngIntElt`

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: Intersection of Two Plane Curves (ex-821adf)

In this example we intersect a pair of plane curves. (Note that much more specialised machinery for working with curves is available in Chapter *Algebraic Curves*.) 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 cannot expect that here, but the final line above does confirm that we have not 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.

122.9 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.

122.9.1 Point Conditions

`IsSingular(p): Pt -> BoolElt`

`IsSingular(X, p): Sch, Pt -> BoolElt`

Returns `true` if and only if the point p is a singular point of the scheme X.

`IsNonsingular(p): Pt -> BoolElt`

`IsNonsingular(X, p): Sch, Pt -> BoolElt`

Returns `true` if and only if the point p is a nonsingular point of the scheme X.

`IsOrdinarySingularity(p): Pt -> BoolElt`

`IsOrdinarySingularity(X, p): Sch, Pt -> BoolElt`

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.

122.9.2 Point Computations

`Multiplicity(p): Pt -> RngIntElt`

`Multiplicity(X, p): Sch, Pt -> RngIntElt`

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): Pt -> Sch`

`TangentSpace(X, p): Sch, Pt -> Sch`

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): Pt -> Sch`

`TangentCone(X, p): Sch, Pt -> Sch`

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.

122.9.3 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): Pt, RngIntElt -> BoolElt, RngMPolElt, SeqEnum, Rec`

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 X is reduced at p - for example, if X is a variety - the second condition follows automatically from the first).

If this is true, the function also returns extra values. The second return value is a multivariate polynomial F_1 in a polynomial ring $k[x_1, \ldots, x_{d+1}]$ such that the singularity p is analytically equivalent to that of the analytic hypersurface $F \in k[[x_1, \ldots, x_{d+1}]]$ 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): Rec, RngIntElt, RngMPol -> RngMPolElt

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): Rec, FldFunRatMElt, RngIntElt, RngMPol -> RngMPolElt, RngMPolElt

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). The argument 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 is not the base ring of X. The argument 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 respect 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 does not vanish on X.

MilnorNumberAnalyticHypersurface(dat): Rec -> RngIntElt

TjurinaNumberAnalyticHypersurface(dat): Rec -> RngIntElt

The record *dat* should be the data record returned for a hypersurface singularity p in $X(k)$ by the first intrinsic of this subsection. It represents a specific power series $F(x_1, \ldots, x_{d+1}) \in k[[x_1, \ldots, x_{d+1}]]$ as described above.

Returns the Milnor (resp. Tjurina) number of this power series F. That is, the k-dimension of the quotient of $k[[x_1, \ldots, x_{d+1}]]$ by the ideal generated by the formal derivatives of F (resp. the formal derivatives of F and F). This may be infinite, in which case `Infinity` is returned. The Tjurina number is infinite if an only if p is *not* an isolated singularity. The same is true for

the Milnor number if the characteristic of k is 0. Note that the Milnor number can depend on the choice of F (F multiplied by any power series with non-zero constant term defines the same analytic hypersurface singularity) whereas the Tjurina number does not - it only depends on the ideal generated by F.

Example: Hypersurface Singularities of a Del Pezzo (ex-aae880)

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 := PointsOverAlgebraicClosure(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 +
    8*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
> // Both singularities are simple $A_1$ singularities which are
> // characterised by having Milnor number 1.
> MilnorNumberAnalyticHypersurface(dat);
1
```

122.10 Classification and Normal Forms of Singularities

This section describes functionality to classify isolated singularities which are analytically hypersurface singularities as defined in the previous subsection *Analytically Hypersurface Singularities*. The classification is up to analytic isomorphism. The singularities are specified by giving a multivariate polynomial or power series f such that the completion of the local ring of the singularity is isomorphic to the quotient by f of the power series ring in which it lies. The analytic classification comes down to finding nice spaces of relatively simple polynomials (the *normal forms*) - these form the classification families - such that any power series with a singularity at the origin is analytically equivalent to one of these normal

forms. The classification type refers to the particular family that the normal form lies in. Elements of distinct families should not be analytically equivalent. The normal forms within a family should depend on a finite number of continuous parameters and at most finitely many forms within the family may be analytically equivalent to any given form (a finite group of family automorphisms is allowable).

The classification scheme we follow here is that of Arnold as described in Chapter 15 of [AGZV85]. For those singularities that lie in one of the families of this classification, we determine the family and normal form. Note that his classification is a partial classification giving a finite set of multi-indexed, irreducible families occurring in the infinite stratification by Milnor number of the space of all isolated analytic hypersurface singularities up to *stable equivalence*. However, his classification at least covers all singularities with Milnor number ≤ 16 and all irreducible families with constant Milnor number that have codimension ≤ 10 in the full space. It includes much more besides: any of his indexed families contain singularities with an infinite range of Milnor numbers as the (integer) index varies.

Any power series in characteristic $\neq 2$ with no constant or linear term can be analytically transformed into a diagonal quadratic part $x_1^2+\ldots+x_r^2$ plus a power series in the remaining variables with all terms of degree ≥ 3. *Stable equivalence* of power series means that the residual degree ≥ 3 parts are analytically isomorphic.The *corank* refers to the number of variables not lying in the quadratic part.

Arnold's families are all *affine, polynomial* families : the normal forms for the family are a fixed polynomial plus a linear combination of a finite number of monomials, all in a subset of the variables and all terms of which are of degree ≥ 3, which is added to the diagonal quadratic form in the remaining variables. The degree ≥ 3 part is the essential part and we sometimes use the term normal form to refer to just this component of the full normal form. It is a polynomial in a number of variables equal to the corank of the singularity.

For example, the normal forms of the corank 2 family denoted $J_{k,0}$ by Arnold with $k > 2$ are the polynomial $f = x^3 + bx^2y^k + y^{3k}$ (b is a constant here) added to an arbitrary linear combination of the $k-2$ monomials $xy^{2k+1}, \ldots, xy^{3k-2}$ and then added to the diagonal quadratic part. So if the number of variables is 3 - let them be x, y, z - we would have $p(x, y) + z^2$, where $p(x, y)$ is f plus the xy^r part. If we have 4 variables x, y, z, t, we get $p(x, y) + z^2 + t^2$ etc. In our implementation, we actually transform to a normal form where the diagonal quadratic part consists of the initial variables and the essential corank part is in the last variables.

As well as computing the normal form, the user can ask for an explicit analytic isomorphism that transforms the original hypersurface equation to the normal form up to terms of any required precisional degree. If the precision is not specified, the transformation is computed up to terms of a sufficiently high degree to uniquely determine the normal form (i.e. the image of the truncated transformation is the normal form plus higher degree error terms that do not affect the analytic equivalence class). In the many cases of families where there is a natural weighting of the variables associated to the normal forms, the default precision may be different for different variables. This will be described more precisely in the intrinsic descriptions.

Arnold's classification is over an algebraically-closed field. Implementationally, the hypersurfaces are generally given over number fields or finite fields. Although we transform and work over finite extensions when absolutely necessary, we try to avoid the overhead of forming too many field extensions. Thus, we often do not normalise completely when computing normal forms, returning an almost-normal form that can easily be transformed to the full normal form by the user if desired. The major way in which we avoid complete normalisation is in not replacing terms of the form ax^r with x^r by taking an rth root of a and transforming by $x \mapsto a^{(-1/r)}x$. These scaling transformations to reduce a coefficient in the main polynomial to 1 are only applied in cases where no roots need be computed (e.g. we can transform axy^3 to xy^3 by mapping x to $(1/a)x$). In particular, the quadratic part of the normal form will be of the form $a_1x_1^2 + \ldots + a_rx_r^2$ for non-zero $a_1, \ldots, a_r$ rather than $x_1^2 + \ldots + x_r^2$. Beside avoiding these scalings, there are a few other places where we use a slightly different normal form to Arnold's in order to avoid field extensions. These will be listed in the relevant intrinsic descriptions.

The classification is over an (algebraically-closed) field of characteristic zero. However, for any particular family, the classification process leading to that family and the transformation to the specific normal forms within that family work in all but a small number of characteristics (the precise ones depending on the family). As it is often useful in computational geometry to work modulo a moderate size prime to avoid coefficient blow-up in Gröbner basis operations, we put no restriction on the characteristic of the base fields over which the singularities are defined. We have tried to be very careful in the implementation

to work out for precisely which characteristics each (indexed) family does not work and only disallow input at the point where we have reached a family and found that we have a bad characteristic for it. So, if the user is working in a positive characteristic the intrinsics may return a runtime error at some stage. The error message indicates the apparent type of the singularity and warns that it is an invalid family for the characteristic. In valid cases, the polynomial normal forms are the same in the positive characteristic as in characteristic zero and the isomorphic transformation to the result can be computed to an arbitrarily high precision (the finite precision transformations returned are the truncations of genuine analytic isomorphisms in that characteristic).

The input hypersurfaces can be given as exact polynomials or defined by the data records returned by the `IsHypersurfaceSingularity` intrinsic of the previous subsection. These records allow the computation to any desired precision of the expansion of a power series F that defines the singularity at the origin. This is sufficient to determine the family and normal form. In fact, if the Milnor number of the power series F is a finite number μ (it is finite for any isolated singularity in characteristic zero and for all of our families in the allowable characteristics) then any other power series that is equal to F up to terms of degree $\mu + 2$ or higher, is (formally) analytically isomorphic to F. This gives a general upper degree bound in terms of Milnor number for the degree that expansion up to will determine the singularity up to isomorphism. Actually, the normal forms of the various families usually have highest degree terms of degree much lower than the Milnor number and we are careful in our implementation to only expand as far as necessary to determine the family, normal form, and truncated transformation.

All of Arnold's families are of corank ≤ 4, with the majority of corank 2 and 3. In the initial Magma V2.21 release, three of the corank 3 families and two of the more complex corank 2 families remain unimplemented. We hope to add the remaining corank 3 cases for a mid-year release.

The Arnold classification is a fundamental result in singularity theory and the functionality to be able to apply it to Magma schemes should be useful in a number of situations. One of our future aims is to use it to speed up the formal desingularisation procedures for the wide range of singular surface models that only contain isolated hypersurface singularities. The normal forms are of a very simple shape - containing only a few monomials in the majority of cases - so the singularities should be much faster to resolve after transformation to normal form.

`NormalFormOfHypersurfaceSingularity(f): RngMPol -> BoolElt, RngMPolElt, MonStgElt, Map`

```
get_trans     : BoolElt                    Default: false
d             : RngIntElt                  Default: 0
fData         : List                       Default: [**]
milnor_number: RngIntElt                   Default: -1
```

The main intrinsic to classify an isolated hypersurface singularity, compute its Arnold normal form, and optionally compute to any desired precision an explicit analytic isomorphism to that form.

The argument f should be a multivariate polynomial over a field k giving the equation of an affine hypersurface with a singularity at the origin $(0, 0, \ldots, 0)$. The singularity does not have to be defined by a global hypersurface. It can be an analytically hypersurface singularity as described in the previous subsection *Analytically Hypersurface Singularities*. To use the intrinsic in this context, the data record $fdat$ returned as the fourth value by `IsHypersurfaceSingularity` is required. Then f should be the expansion to some finite degree $fdeg$ of the power series F defining the singularity (i.e. $F = f+$ terms of degree $\geq fdeg + 1$ where f should not be 0 here!) and parameter `fData` should be set to the two-element Magma list `[* fdat, fdeg *]`. The element $fdat$ will be used to expand f to higher precision if necessary. The intrinsic makes use of the Milnor number of the singularity. If this is already known, the user can pass it in using the parameter `milnor_number`. This can avoid an expensive recomputation for more complicated singularities. Note, however, that the wrong value passed in will result in runtime errors or invalid return values.

If an explicit analytic isomorphism ϕ from f to the normal form is also required, the parameter `get_trans` should be set to `true`. Let the variables of f be $x_1, \ldots, x_n$ and let f_0 denote the normal form. Then there is a sequence of n power series in $R = k[[x_1 \ldots, x_n]]$ with constant term zero, $[s_1 \ldots, s_n]$, such that ϕ is an isomorphism $R \to R$, $x_i \mapsto s_i$, mapping f to f_0. The condition that ϕ is an isomorphism is equivalent to the matrix of coefficients of the degree 1 terms of the s_i being invertible. When f_0 is in $L[x_1, \ldots, x_n]$ for a finite field extension L of k, ϕ is defined by power

series over L. Of course, we cannot represent the power series of the isomorphism exactly, so ϕ is represented by a Magma homomorphism from $L[x_1, .., x_n]$ (the base change of the parent ring of f to L) to itself which is defined by the sequence of polynomials p_i which are the s_i truncated to a sufficiently high degree.

The precision of the map representing ϕ is determined as follows. The default precision is the smallest to determine f_0 uniquely. The truncated polynomial map defined by the p_i maps f to f_0+e, with an error e. The precision of the map is such that e consists of terms of degree higher than those of f_0 with respect to a natural weighting of the variables and such that $f_0 + e$ is analytically isomorphic to f_0 and no other normal form in the same (or different) family. Many of the families in the classification have *semiquasihomogeneous* normal forms in Arnold's terminology. This means that there is a positive (integer) weighting for the variables for which the fixed polynomial part g of the normal forms (see the introductory section above) is homogeneous (and non-degenerate) of weighted degree d_0 and the variable part (if it exists) is a linear combination of monomials of higher degree. In these cases, the default precision is the highest possible weighted degree of the monomials in the variable part (or d_0 if there is no variable part). What this means is that there might be much lower ordinary degree terms in the transformation error e for higher weight variables. This convention may seem a little awkward at first but it seems to be the most natural thing in terms of the way that these cases are processed by the code. For non-semiquasihomogeneous cases or when parameter `d` is used (see below), the precision is always in terms of the ordinary degree of monomials.

If parameter `d` is set to a positive value (its default is 0), ϕ will be computed to a precision (in terms of ordinary degree) which is the maximum of d and the default (ordinary) degree needed to determine the normal form uniquely. That is, all terms in the error e are guaranteed to be of ordinary degree greater than d.

The first return value is `true` if the singularity lies in one of the families of Arnold's classification and `false` otherwise. If `true`, the second return value is the normal form f_0 of f and the third is a string giving the name of the family. If parameter `get_trans` is `true`, then a fourth return value is the polynomial map giving the (truncated) isomorphism from f to f_0 as described above.

As discussed in the introduction, the base field k can be of arbitrary characteristic. The only restriction on its Magma type is that it must be a field for which finite separable extensions are constructible. A runtime error will occur if it turns out that the shape of the singularity is of a family type for which the characteristic of k is invalid. The apparent family type will be given in the error message. Characteristic 2 will lead to an error in virtually (but not quite) all cases. In particular, an error will result for characteristic 2 fields whenever f has a non-trivial quadratic part. If the singularity is not isolated, this will be discovered when the Milnor number of f is computed (it will be infinite) and `false` will be returned.

We allow a non-singular point, which we class as a 'singularity' of type A_0 for which the normal form is x_1.

All of the families in the version of Arnold's classification that we use have corank ≤ 4 (see intro). The only corank 0 singularity is the A_1 singularity whose normal form is $x_1^2 + \ldots + x_n^2$. As discussed in the introduction, to avoid excessive field extensions, we do not scale the variables in our transformations to normal form in order to make coefficients 1, so the normal form returned will actually be $a_1x_1^2 + \ldots + a_nx_n^2$ for some non-zero constants a_i.

The corank 1 singularities are the $A_r, r \geq 2$. The description strings for these are "A2", "A3" etc. The normal form for these are $a_1x_1^2 + \ldots + a_{n-1}x_{n-1}^2 + a_nx_n^{r+1}$ where, again, we do not scale the variables to make all of the a_i equal to 1.

Corank 2 singularities are described below for the `Corank2Case` intrinsic. That intrinsic is used by this one but it has also been made available for the user to call directly. There are 3 families of corank 2 singularities that have not yet been covered.

Code for corank 3 in the initial release of V2.21 does not cover all of the families, although we hope to complete it in a mid-year release. The families not yet included lead to a "not yet available" runtime error.

The only corank 4 case in the classification (sometime not included) is O_{16} (descriptor: "O16").

The normal form for these is the diagonal quadratic part added to a non-degenerate homogeneous cubic in 4 variables (non-degenerate meaning that it defines a non-singular projective cubic surface) and a 1-dimensional 'variable' part (maybe zero) given by a homogeneous degree 4 polynomial in the 4 variables. The standard normalisation here is to transform the homogeneous cubic to the form $x^3 + y^3 + z^3 + t^3 + (ax + by + cz + dt)^3$ for constants a, b, c, d and the variable part is then given by a scaler multiple of the monomial $xyzt$. Again, to avoid a possibly large field extension and because not all non-singular cubics can be linearly transformed into this form, we currently take a lazy approach and do not attempt to normalise the cubic part into any particular shape. If this case occurs and the user really wants to try to get a transformation to the standard form, he may use intrinsic *PentahedronIdeal* to construct a linear transformation in the last 4 variables to move the cubic part to the correct shape. Applying this to the full normal form returned will give the Arnold normal form.

The implementation follows the decision procedure laid out in Chapter 16 of [AGZV85]. The transformations to full normal form given the lowest degree part are effected in a uniform manner via 3 functions that respectively deal with the semiquasihomogeneous cases, the non-sqh cases that satisfy condition A for a Newton filtration (see page 212 of [AGZV85]), and finally the non sqh-cases for which the spectral sequence analysis of Chapter 14 of [AGZV85] applies. In many of the non-sqh cases, the sqh function is first applied to get to the fixed part of the normal form and then the relevant other function is used to find the variable part and kill off other terms.

`Corank2Case(f): RngMPol -> BoolElt, RngMPolElt, MonStgElt, Map`

```
get_trans    : BoolElt                    Default: false
milnor_number: RngIntElt                  Default: -1
d            : RngIntElt                  Default: 0
fData        : List                       Default: [**]
```

This is a direct version of the previous general intrinsic that only deals with the non-quadratic part of hypersurface singularities of corank 2 as defined in the introduction. Argument f should be a polynomial in two variables with all terms of degree at least 3 which is a defining equation for the singularity at the origin $(0, 0)$. The parameter `milnor_number` if given a positive value can be used to input the known Milnor number of f to save a small amount of computation time. The remaining parameters and return values have the same meaning as for the general intrinsic. In particular `fData` can be used when f is the truncation of a power series F that gives the exact defining equation for the (analytically) hypersurface singularity.

The families of singularities covered by this case are, in Arnold's notation: D_k, $k \geq 4$; $E_{6k}, E_{6k+1}, E_{6k+2}$, $k \geq 1$; $J_{k,i}$, $k \geq 2, i \geq 0$; $X_{k,i}$, $k, i \geq 1$; $Y^1_{r,s}$, $r \geq s \geq 1$; $Z_{6k+5}, Z_{6k+6}, Z_{6k+7}$, $k \geq 2$; $Z_{k,i}$, $k \geq 1, i \geq 0$; $W_{12k}, W_{12k+1}, W_{12k+5}, W_{12k+6}$, $k \geq 1$; $W_{k,i}$, $k \geq 1, i \geq 0$; $W^{\#}_{k,i}$, $k \geq 1, i \geq 1$; N_{16}.

The description strings are the obvious translations of these: e.g., for $Y^1_{2,1}$, we return `"Y^1_{2, 1}"`. There are 3 multi-indexed families with atypical product normal forms that we have not yet finished implementing: $Y^k_{r,s}, Z^k_r, Z^k_{i,j}$ for $k \geq 2$. Currently, if the analysis leads to one of these families, a runtime error results informing the user of the precise family type but warning that the exact normal form (and transformation) cannot currently be computed.

The expressions for normal forms over an algebraically-closed field for each family can be found in Chapter 15 of [AGZV85]. As stated in the introduction, in order to avoid unnecessary field extensions, the normal forms we return are either not fully normalised or are slightly different to those given there. Besides the ubiquitous avoidance of scaling a variable x to transform a term of the form ax^r to x^r with $r \geq 2$, the following variants are used.

For $J_{k,0}$, $k \geq 2$, instead of taking the quasihomogeneous part of the normal form to be $ax^3 + bx^2y^k + cy^{3k}$, we instead use $ax^3 + bxy^{2k} + cy^{3k}$ $(a, b, c \neq 0)$ except in the one case when transformation to this by $x \mapsto x + uy^k$ leads to $c = 0$. Then we use the usual form $ax^3 + bx^2y^k + cy^{3k}$ with $2b^3 = 27a^2c$ (this gives one isomorphism class over the algebraic closure on scaling x and y appropriately). The monomial basis for the variable part of the normal form remains $xy^{2k+1}, \ldots, xy^{3k-2}$.

Similarly, for $Z_{k,0}$, $k \geq 1$, instead of using $y(ax^3 + bx^2y^{k+1}k + cy^{3k+3})$ for the quasihomogeneous

part, we use $y(ax^3 + bxy^{2k+2}k + cy^{3k+3})$ except in the same unique case as above. The monomial basis for the variable part of the normal form remains $xy^{2k+4}, \ldots, xy^{3k+3}$.

Singularities of type N_{16} have a normal form which is a homogeneous polynomial of degree 5 in x, y that is non-degenerate (i.e. it splits into distinct linear factors over the algebraic closure of the base field) plus a higher degree part which lies in a one-dimensional subspace of homogeneous polynomials of degree 6. The normal form given by Arnold over an algebraically-closed field involves a linear transformation of the homogeneous part to a polynomial of the form $x^4y + ax^3y^2 + bx^2y^3 + xy^4$, in which case the full normal form is this degree 5 polynomial plus a term of the form cx^3y^3. As the transformation of the degree 5 part generally involves a field extension of degree 20 (2 roots of a degree 5 polynomial need to be added to the base field), we do not make this transformation. We leave the homogeneous degree 5 part f_5 as it first occurs and the full normal form returned will be of the form $f_5 + f_6$ where f_6 is a homogeneous degree 6 part that lies in a one-dimensional subspace, depending on f_5, of degree 6 x, y-polynomials. As for the O_{16} case in the general intrinsic, the user may determine the Arnold normal form by finding the linear translation on f_5 to map it to standard form and apply this transformation to $f_5 + f_6$.

`Corank3Case(f): RngMPol -> BoolElt, RngMPolElt, MonStgElt, Map`

```
get_trans    : BoolElt                    Default: false
milnor_number: RngIntElt                  Default: -1
d            : RngIntElt                  Default: 0
fData        : List                       Default: [**]
```

This is the direct version of the general `NormalFormOfHypersurfaceSingularity` intrinsic that only deals with the non-quadratic part of hypersurface singularities of corank 3 as defined in the introduction. Argument f should be a polynomial in three variables with all terms of degree at least 3 which is a defining equation for the singularity at the origin $(0, 0, 0)$. The parameters are exactly the same as for the above `Corank2` intrinsic.

The families of singularities covered by this case are, in Arnold's notation: $P_k = T_{3,3,k-5}$, $k \geq 8$; $R_{p,q} = T_{3,p,q}$, $4 \leq p \leq q$; $T_{p,q,r}$, $4 \leq p \leq q \leq r$; $Q_{6k+4}, Q_{6k+5}, Q_{6k+6}$, $k \geq 1$; $Q_{k,i}$, $k \geq 2, i \geq 0$; $S_{12k-1}, S_{12k}, S_{12k+4}, S_{12k+5}$, $k \geq 1$; $S_{k,i}$, $k \geq 1, i \geq 0$; $S^{\#}_{k,i}$, $k, i \geq 1$; U_{12k}, U_{12k+4}, $k \geq 1$; $U_{k,i}$, $k \geq 1, i \geq 0$; $V_{1,i}, V^{\#}_{1,i}$, $i \geq 0$.

Again, the description strings contained in the third return value are obvious text translations of these. In the first release covering corank 3 singularities (V2.21, initial release), some of the later families have not yet been covered and these will give a "Not yet available" runtime error. These cases should be implemented and exported in a mid-year release. We do not cover the S^*, U^* and V^* families that Arnold gives as particular classes with specific invariants but without precise normal forms. A singularity falling into one of these classes will result in a first return value of `false`.

Apart from the usual avoidance of root-scaling variables to transform, e.g., ax^r to x^r when $r \geq 2$, the only major difference between the normal forms computed and Arnold's versions is for P_8. A P_8 singularity is one whose equation can be transformed into a homogeneous cubic (in 3 variables) that is non-singular as the equation for a projective (genus 1) curve. Arnold uses the normal form $x^3 + y^3 + z^3 + axyz$ where a is the family parameter. We transform to a non-singular cubic but do not convert it to this specific form, partly to avoid the necessary field extensions. the user may use Magma functions for elliptic curves to find the explicit transformation to Arnold's form. Two P_8 singularities given by homogeneous cubics are isomorphic if and only if the elliptic j-invariants are the same, where each cubic is considered as an elliptic curve by choosing any projective point on it as the origin. If the chosen origin is actually one of the 9 inflexion points of the cubic then an elliptic curve isomorphism between the two will be a linear transformation in the base variables which therefore gives an analytic isomorphism between the two singularities.

Example: Normal Forms of Isolated Singularities (ex-41d136)

We give some examples of the analysis of isolated singularities. The first is a corank 2 E_{13} singularity defined by a 3-variable polynomial (at the origin). For these singularities, the non-quadratic part

(in variables y and z) of the (unscaled) normal form is $ay^3 + byz^5$ added to a cz^8 term where c may be zero.

```
> Q := Rationals();
> P<x,y,z> := PolynomialRing(Q,3,"grevlex");
> f := z^8-3*z^7-3*x*z^5+3*y*z^5+z^6+3*x*z^4-3*y*z^4+2*y^4+3*x^2*z^2-6*x*y*z^2+
>   3*y^2*z^2+x^3-3*x^2*y+7*x*y^2-y^3-4*y^2*z+2*x^2-4*x*z+2*z^2;
> boo,f0,typ := NormalFormOfHypersurfaceSingularity(f);
> boo; f0; typ;
true
z^8 + 3*y*z^5 - y^3 + 2*x^2
E13
```

The normal form has highest term of degree 8. Now, call the intrinsic again to get the transformation that takes f to f_0 up to terms of degree at least 9.

```
> _,_,_,mp := NormalFormOfHypersurfaceSingularity(f : get_trans := true, d := 8);
> mp_err := mp(f)-f0;
> Min([TotalDegree(t) : t in Terms(mp_err)]);
9 5440
```

For the second example, we take the singularity defined by a two-variable polynomial containing no terms of degree ≤ 3. Here, we can use the `Corank2Case` intrinsic directly. The singularity is of type $Z_{1,1}$. Although the polynomial may define an irreducible curve (as in this example), the singularity is non-normal and analytically reducible. The defining polynomial splits into a product of 2 irreducible factors in the power series ring. This can be seen from the normal form which is divisible by the linear factor y.

```
> P<x,y> := PolynomialRing(Rationals(),2,"grevlex");
> f :=  y^12-6*x*y^10+15*x^2*y^8+x^9+9*x^8*y+36*x^7*y^2+84*x^6*y^3+
>  126*x^5*y^4+126*x^4*y^5+64*x^3*y^6+36*x^2*y^7+9*x*y^8+y^9+2*x^8+
>  16*x^7*y+56*x^6*y^2+112*x^5*y^3+155*x^4*y^4+112*x^3*y^5+56*x^2*y^6+
>  16*x*y^7+2*y^8-6*x^5*y^2-x^3*y^4-3*x^2*y^5-4*x*y^6-2*y^7+x^6+2*x^4*y^2+
>  6*x^3*y^3+9*x^2*y^4+5*x*y^5-x^5-3*x^4*y-6*x^3*y^2-4*x^2*y^3+x^4+x^3*y;
> IsIrreducible(f);
true
> boo,f0,typ := Corank2Case(f);
> boo; f0; typ;
true
y^9 + 2*y^8 - x^2*y^3 + x^3*y
Z_{1,1}
```

As a final example, we consider an E_7 singularity p on a degree 6 complete intersection surface in $\mathbf{P}^4$. Here, we cannot find a polynomial giving a global hypersurface defining p by simple elimination of a variable. However, p is *analytically* a hypersurface singularity (as all E_7 singularities are) and we use the `IsHypersurfaceSingularity` intrinsic to give f, the truncation to degree 3 of a power series F defining p analytically, and the $fdat$ data object that allows expansion of F to higher degree.

Calling the main intrinsic with just f as an argument leads to a negative result (in fact, it is clear by inspection that f has a non-isolated singularity at the origin). A second call with the `fData` parameter included, however, leads to the correct result.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Q,4);
> X := Scheme(P4,[x^2+y*t+t^2,z^3+x*t*u+y^2*u-t*u^2]);
> p := X![0,0,0,0,1];
> _,f,_,fdat := IsHypersurfaceSingularity(p,3);
> R<a,b,c> := Parent(f);
> f;
```

(continues on next page)

(continued from previous page)

```
b^3 - c^3 + c^2
> NormalFormOfHypersurfaceSingularity(f);
false
> boo,f0,typ := NormalFormOfHypersurfaceSingularity(f : fData := [*fdat,3*]);
> boo; f0; typ;
true
b*c^3 + b^3 + a^2
E7
```

122.11 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): Sch -> RngIntElt`

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): Sch -> RngIntElt`

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): Sch -> RngIntElt`

The degree of the scheme X.

`ArithmeticGenus(X): Sch -> RngIntElt`

The arithmetic genus of a scheme X. The ambient space of X must be ordinary projective space.

`IsEmpty(X): Sch -> BoolElt`

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): Sch -> BoolElt`

```
FullCheck: BoolElt                    Default: false
```

Returns `true` if and only if the scheme X is nonsingular over an algebraic closure of its base field (i.e. smooth) where X should be equidimensional if parameter `FullCheck` is `false`, which is the default. 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. If X is reducible with components of different dimensions that do not intersect, this test may not pick up singularities of smaller dimensional components, since it finds the points where the tangent space has larger dimension than the dimension of X (= the dimension of the largest component).

If `FullCheck` is set to `true`, then the test will also work if the scheme is not equidimensional. However, this involves a potentially expensive decomposition of X into equidimensional components, with the Jacobian criterion being applied to each one if no two intersect.

This smoothness test is a valid test for affine schemes or ones in ordinary or product projective space. For other ambients, the Jacobian test applied this way will not necessarily give correct results for the closed subset of X where it intersects singularities of the ambient space.

For X in a weighted projective ambient (i.e., one with a single grading consisting of positive integer weights), we have now extended the non-singularity test to the full scheme using `WeightedAffinePatch`. The bad subscheme which intersects the singularities of the ambient is computed and the test is done along this by pulling over to appropriate weighted affine patches. As these patches can be in quite high-dimensional ambients with a number of additional equations, this can be quite a heavy computation in general. We have added a number of speed-ups, computing the equations of X in the patch by a simplified method, where appropriate, and also by performing the generally much faster point singularity test after decomposing a cluster and base-field extending when the bad subscheme has zero-dimensional components. There is a singular K3-surface example below where the computations are very slow without these speed-ups.

Once computed, the result is stored in the attribute `X`Nonsingular`. If the user knows *a priori* that $X$ is singular/non-singular, they may directly set the `X`Nonsingular` attribute to `true` or `false` as appropriate. This can save a lot of time, avoiding the smoothness test having to be carried out if other intrinsics are applied to X that need the result.

`IsSingular(X): Sch -> BoolElt`

```
FullCheck: BoolElt                    Default: false
```

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. This is just the negation of the previous intrinsic and the `FullCheck` parameter and extra functionality for weighted projective space is the same as for that.

`JacobianSubrankScheme(X): Sch -> Sch`

The subscheme of the scheme X defined by the vanishing of the minors of the Jacobian matrix of X of size the codimension of X in its ambient. For X affine or ordinary or product projective, this defines the set of points where the dimension of the tangent space is greater than the dimension of X. If all irreducible components of X are of the same dimension, this gives the non-smooth subscheme of X and is used by `SingularSubscheme`.

`SingularSubscheme(X): Sch -> Sch`

```
FullCheck: BoolElt                    Default: false
```

Scheme X should be affine or ordinary or product projective. Returns the non-smooth subscheme of X. If the `FullCheck` parameter is `false`, this just computes `JacobianSubrankScheme` and will contain all lower-dimensional components of X if X is not equidimensional, so may contain smooth points of those. If `FullCheck` is set to true, X is first decomposed into equidimensional components and the result is computed as the union of the non-smooth subschemes of these along with intersections between components. As with the singularity test intrinsics, the decomposition can take a lot of time.

Once computed, the result is stored in the `X`SingularSubscheme` attribute. If for some reason, the user knows the result beforehand, or has computed it independently for $X$ in an ambient for which this intrinsic is invalid, they may set the `X`SingularSubscheme` attribute directly to the relevant subscheme of X (note: the assigned subscheme should have X as its `SuperScheme`). This can be useful for later calls to intrinsics that require the singular subscheme, like checking for simple singularities or desingularising surfaces. If the user does assign this attribute directly, they should also set the `X`Nonsingular` attribute appropriately, if that has not already been assigned.

Example: Singularities in Weighted Projective Space (ex-677f5d)

In this example we give two examples of full singularity checks in weighted projective space. The first is a singular K3-surface from the K3-database that is in a highly-weighted space $\mathbf{P}(4,5,6,15,10)$. The full check is very slow without the simplifications, but the bad subscheme which requires additional checking via weighted patches is zero-dimensional and the reduction to point singularity checks really speeds things up.

```
> Q := Rationals();
> P<x,y,z,t,u> := ProjectiveSpace(Q,[4,5,6,15,10]);
> X := Scheme(P,[2*x^5*y^2 + 3*x^6*z + y^6 + 4*x*y^4*z + 3*x^2*y^2*z^2 +
>     x^3*z^3 + 5*x^5*u + z^5 + y^4*u + 3*x*y^2*z*u +
>     3*x^2*z^2*u + 3*y^3*t + 2*x*y*z*t + 3*y^2*u^2 +
>     5*x*z*u^2 + 3*y*t*u + 4*u^3 + 3*t^2,
> 4*y^2 + x*z]);
> time IsNonsingular(X);
false
Time: 0.150
```

The second example uses the rather artificial $\mathbf{P}(1,2,2)$ weighted plane. If x is the first variable, we take the subscheme X defined by x^2. The weighted plane is isomorphic to the ordinary projective plane in such a way that X corresponds to the line $x = 0$. So, clearly, X is non-singular but the straight Jacobian criterion goes horribly wrong here, as X looks like a doubled line which is everywhere singular if the gradings are not taken into account. In this case, the "bad" subscheme of X at which extra checks need to be performed is the whole of X!

```
> P<x,y,z> := ProjectiveSpace(Q,[1,2,2]);
> X := Scheme(P,x^2);
> IsNonsingular(X);
true
```

PrimeComponents(X): Sch -> SeqEnum

A sequence containing the irredundant prime components of the scheme X, i.e. reduced and irreducible subschemes of X.

PrimaryComponents(X): Sch -> SeqEnum

A sequence containing the irredundant primary components of the scheme X, i.e. irreducible (but not necessarily reduced) subschemes of X.

IrreducibleComponents(X): Sch -> SeqEnum

This returns the same as PrimaryComponents.

ReducedSubscheme(X): Sch -> Sch, MapSch

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): Sch -> BoolElt

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): Sch -> BoolElt

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): Sch -> BoolElt

IsGorenstein(X): Sch -> BoolElt

IsArithmeticallyCohenMacaulay(X): Sch -> BoolElt

IsArithmeticallyGorenstein(X): Sch -> BoolElt

```
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: Prime Components of a Scheme (ex-5b784a)

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.

122.12 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 $m\colon k \to L$ 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): Sch, Rng -> Sch`

`BaseExtend(A, K): Sch, Rng -> Sch`

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): Sch, Map -> Sch`

`BaseExtend(A, m): Sch, Map -> Sch`

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): SeqEnum, Rng -> SeqEnum`

`BaseExtend(F, K): SeqEnum, Rng -> SeqEnum`

`BaseChange(F, m): SeqEnum, Rng -> SeqEnum`

`BaseExtend(F, m): SeqEnum, Rng -> SeqEnum`

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): Sch, Sch -> Sch`

`BaseExtend(X, A): Sch, Sch -> Sch`

`BaseChange(X, A, m): Sch, Sch, Map -> Sch`

`BaseExtend(X, A, m): Sch, Sch, Map -> Sch`

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): Sch, RngIntElt -> Sch`

`BaseExtend(X, n): Sch, RngIntElt -> Sch`

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: Intersection Points after Base Change `(ex-bc94d7)`

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
]
```

122.13 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.

There are also nonstandard affine patches for weighted projective spaces for indices where the standard affine patch is undefined. These are necessarily more complicated than the standard affine patches, which are plain affine ambients, and as such, cannot be integrated into the lower level scheme machinery in the way that the standard patches are. However, they can be a very useful user tool for local computations.

`ProjectiveClosure(X): Sch -> Sch`

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 *Closure of a Degenerate Patch*).

`AffinePatch(X, i): Sch, RngIntElt -> Sch`

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 $1, \ldots, n + 1$.

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): Sch, Pt -> Sch, Pt`

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): Aff -> BoolElt, RngIntElt`

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): Sch -> BoolElt`

Return the number of standard affine patches of the scheme X (O if X is an affine scheme).

`HasAffinePatch(X, i): Sch, RngIntElt -> BoolElt`

Return whether the ith patch of the scheme X can be created.

`WeightedAffinePatch(P, i): Prj, RngIntElt -> Sch, MapIsoSch`

Computes and returns an "affine patch" A of a weighted projective ambient P for any index i along with the projective closure map ϕ from A to P. Here, weighted projective space means a projective space with a single grading where all weights are positive integers. If the number of coordinate variables is n, the index i can be any number between 1 and n and it indexes patches in the usual manner. If the corresponding weight is 1 (i.e. the $(n+1-i)$th entry of the unique grading array is 1), then the standard affine patch is returned along with its usual projective closure map. Here, A is an $(n-1)$-dimensional affine ambient and ϕ is an everywhere defined inclusion map on A.

When the $(n+1-i)$th weight is > 1, A is an affine scheme, but it will generally have defining equations and its affine ambient space will have more than $n-1$ variables, except in the case that the weight divides all other weights. The projective closure map ϕ here will have multiple sets of alternative defining polynomials but the combined domain of definition of these will never cover all of A (except in the special case of weight division again). That is, ϕ will have a non-empty base scheme, which in fact will always contain the origin of A. This is unavoidable, using Magma scheme maps. To define the map everywhere requires something like the radical maps of the Toric Geometry package. However, we do compute a sufficient number of alternative sets of defining equations so that the base scheme only contains points where it is impossible to define ϕ by scheme maps. The inverse equations for the map $P \mapsto A$ is defined on the full theoretical domain of definition - that is, at all points in the Zariski-open subset A. Note also that, although the dimension of the affine ambient of the constructed A is often much greater than $n-1$, it is the minimal dimension for affine ambients into which the abstract scheme-theoretic patch may be embedded. This can be seen from the fact that the tangent space at the origin of A is always the full ambient.

Because A is not a plain affine ambient at weight > 1 indices and ϕ is not everywhere defined on A, it cannot be used by the lower-level scheme machinery in the way that standard affine patches are. In particular, it is not stored, it cannot be obtained from the usual `AffinePatch` intrinsic and `HasAffinePatch` will still return `false` for that index. It can be a very useful object for local computations that should be considered as existing at a higher user level and not fitting into the basic scheme machinery in the way that the existing standard affine patches do. We have only defined these weighted patches on the ambients. For subschemes of weighted projective ambients, the corresponding patches can be constructed by the user by pulling across defining equations via ϕ or its inverse. This can be very slow for high weight spaces, unlike for standard patches, and there are simplifications in various cases, which is why we have left it to the user rather than adding an inefficient general function.

Example: Repeated Patches and Closures `(ex-3bb4da)`

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: Closure of a Degenerate Patch (ex-92843d)

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
```

Example: Patches of Weighted Projective Space (ex-dca912)

The first example is the first affine patch of the 3-dimensional weighted projective space $\mathbf{P}(2,3,4,5)$. We see that the affine patch is embedded in an 8-dimensional affine ambient and the projective closure map ϕ is not defined at the origin. Although the patches in this case seems like complicated affine schemes, it is still better to work with them than globally embedding P into 418-dimensional ordinary projective space via the usual embedding.

```
> P := ProjectiveSpace(Rationals(),[2,3,4,5]);
> A,phi := WeightedAffinePatch(P,1);
> A;
Scheme over Rational Field defined by
$.4^2 - $.2*$.6,
$.1^2*$.4 - $.3*$.6,
$.2*$.3 - $.1*$.5,
$.1^2*$.2 - $.3*$.4,
-$.2*$.5 + $.1*$.8,
$.1*$.2*$.4 - $.5*$.6,
$.1*$.3^2 - $.2*$.7,
$.1^3*$.3 - $.4*$.7,
$.1*$.2^2 - $.4*$.5,
$.1^5 - $.6*$.7,
-$.5^2 + $.3*$.8,
$.2^2*$.4 - $.6*$.8,
$.3^3 - $.5*$.7,
```

(continues on next page)

(continued from previous page)

```
$.2^3 - $.4*$.8,
-$.3^2*$.5 + $.7*$.8
> phi;
Mapping from: Sch: A to Prj: P
with equations :
$.6
$.1*$.6
$.4*$.6
$.6^2
and inverse
$.1*$.2/$.4
$.1*$.3^2/$.4^2
$.2^2*$.3/$.4^2
$.1^3*$.3/$.4^2
$.2*$.3^3/$.4^3
$.1^5/$.4^2
$.2^5/$.4^3
$.3^5/$.4^4
and alternative equations :
$.1*$.7
$.7^2
$.3*$.7^2
$.7^3

$.2
$.5
$.8
$.8
> bs := ReducedSubscheme(BaseScheme(phi));
> Dimension(bs); Degree(bs);
0
1
> Support(bs);
{ (0, 0, 0, 0, 0, 0, 0, 0) }
```

The second example shows why it's a good idea to avoid working with an unnecessary weighted projective space. We take the 2-dimensional weighted projective plane $\mathbf{P}(1,2,2)$ which is easily seen to be isomorphic to the ordinary weighted projective plane under $[x:y:z] \mapsto [x^2:y:z]$. However, an everywhere defined set of inverse equations cannot be written down via Magma scheme maps (we'd need radical maps) and that shows up in the projective closure map ϕ from the first affine patch being undefinable along a coordinate axis, even though this patch is just the affine plane, corresponding to the first standard affine patch of the ordinary projective plane.

```
> P := ProjectiveSpace(Rationals(),[1,2,2]);
> A,phi := WeightedAffinePatch(P,1);
> A;
Affine Space of dimension 2 over Rational Field
Variables: $.1, $.2
> phi;
Mapping from: Aff: A to Prj: P
with equations :
$.2
$.1*$.2
$.2
and inverse
$.2/$.3
```

(continues on next page)

(continued from previous page)

```
$.1^2/$.3
> GroebnerBasis(BaseScheme(phi));
[
    $.2
]
```

`HyperplaneAtInfinity(X): Sch -> Sch`

The hyperplane complement of the scheme X in its projective closure.

`ProjectiveClosureMap(A): Aff -> MapSch`

`PCMap(A): Aff -> MapSch`

The map from the affine space A to its projective closure.

`AffineDecomposition(P): Prj -> [MapSch], Pt`

Projective spaces have a standard disjoint decomposition into affine pieces—not the same thing as the affine patches—of the form

$$\mathbf{P}^n = \mathbb{A}^n \cup \mathbb{A}^{n-1} \cup \cdots \cup \mathbb{A}^1 \cup p$$

where $\mathbb{A}^n$ is the first affine patch, $\mathbb{A}^{n-1}$ 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): Sch, Pt -> Sch, MapSch`

An affine patch of S centred at the point p and the embedding into S, achieved by translation of a standard affine patch.

122.14 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.

122.14.1 Height

`HeightOnAmbient(P): Pt -> RngElt`

```
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 or a number field.

122.14.2 Restriction of Scalars

`RestrictionOfScalars(S, F): Sch, Fld -> Sch, MapSch, UserProgram, Map`

`WeilRestriction(S, F): Sch, Fld -> Sch, MapSch, UserProgram, Map`

```
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 $S_{res}(R) \to S(RK)$ (if there is no known relationship between R and K an error results), and
- a map of point sets $S(K) \to S_{res}(F)$.

The result is obtained by direct substitution using the standard basis of K/F, or the basis given in ExtensionBasis if this is specified.

122.14.3 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(K, P): FldAlg, RngOrdIdl → FldLoc, Map`). There is also a version for K a 1-dimensional function field, which checks whether there are points defined over the completion of K at a specified place.

`IsEmpty(Xm): SetPt -> BoolElt, Pt`

```
Smooth           : BoolElt                    Default: false
AssumeIrreducible: BoolElt                    Default: false
AssumeNonsingular: BoolElt                    Default: false
LocSol           : Verbose                    Default: 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 [Bru].

Example: Emptiness of p-adic Point Sets (ex-5c78a9)

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: Empty p-adic Points of a Hyperelliptic Curve (ex-aa0142)

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): Sch, RngOrdIdl -> BoolElt, Pt`

`IsLocallySolvable(X, p): Sch, RngIntElt -> BoolElt, Pt`

```
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 [Bru].

`IsLocallySolvable(X, pl): Sch, PlcFunElt -> BoolElt, Pt`

```
completion_map: Any                    Default: 0
```

Given an affine or projective scheme X of dimension at least 1 and defined over an algebraic function field K, and a place pl of K, returns whether X contains a point defined over the completion of K at pl. If the scheme is found to have such local points, then `true` is returned together with an approximation to a point. Otherwise, `false` is returned.

A current restriction is that K must be a simple extension of a (1-D) rational function field. The completion of K and map m from K to that completion are computed. For speed, if the user has already computed this completion data, the completion map m can be passed in via the parameter `completion_map`. Otherwise the parameter is given the default value 0.

Example: Testing Local Solvability (ex-ca0503)

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): Pt, RngIntElt -> Pt`

```
Strict: BoolElt                    Default: true
```

`LiftPoint(F, d, P, n): [RngMPolElt], RngIntElt, [FldLocElt], RngIntElt -> SeqEnum`

```
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: Lifting a p-adic Point (ex-ed4659)

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))
```

122.14.4 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): Sch[FldRat], RngIntElt -> SeqEnum`

```
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: Searching for Points on a Surface (ex-a468fc)

```
> 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
```

122.14.5 Reduction Mod p

`Reduction(X, p): Sch, Any -> Sch`

```
ambient: Sch                    Default: false
```

X should be an ordinary projective scheme defined over a field K which should be the rational field or a number field. If K is the rationals then argument p should be a rational prime and if K is a number field, p should be a place or a prime ideal in an order of K.

The intrinsic computes and returns the reduction mod p of the *flat closure* of X in projective space over the local ring of p. This reduction is a scheme in projective space of dimension n over the residue class field k_p of p where n is the dimension of the projective ambient of X.

It is possible to enter the ambient that the user wishes the result to lie in using the `ambient` parameter. The value entered should be an ordinary projective space of dimension n over k, a finite field equal to k_p.

Note that the flat closure essentially computes a saturation of the ideal I of X with respect to a local parameter π of p and takes the reduction of this mod p to give the result (though usually a full colon ideal computation is unnecessary, and a simple reduction of basis elements is checked to give the correct result via Hilbert polynomials). This reduction is guaranteed to be of the same dimension as X and have the same Hilbert polynomial, but it is not "minimised" in any way. For example, it will not necessarily give the reduction of the minimal model of a curve - even an elliptic curve.

`Reduction(I, p): RngMPol, Any -> RngMPol`

`parent: RngMPol` Default: `false`

This is a simple variant on the first intrinsic, included for convenience in case the user prefers working with ideals. I should be a homogeneous ideal in a multivariate polynomial ring over K, the rational field or a number field. p is as before. Returns the defining ideal of the reduction mod p of the projective scheme defined by I.

More precisely, if $K[x_1, \ldots, x_n]$ is the polynomial ring in which I lies, the result lies in $k_p[x_1, \ldots, x_n]$, where k_p is the residue class field of p, and can be defined as the reduction mod p of $I \cap O_p[x_1, \ldots, x_n]$, where O_p is the local ring of p.

The user may specify the polynomial ring in which the result is to lie by using the `parent` parameter which should be set to a multivariate polynomial ring in n variables over a finite field k equal to k_p.

Example: Reducing a Scheme Modulo p (ex-cbccd6)

This easy example is a complete intersection of 2 quadric equations in $\mathbf{P}^3$ over the rationals which defines a scheme that decomposes into a union of 4 lines over the algebraic closure. Simply reducing the two given defining quadrics mod 3 gives a 2-dimensional scheme defined by a single equation which is not the reduction of the flat closure. The intrinsic gives the correct result (which isn't hard to find by inspection in this case).

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> S := Scheme (P, [3*x^2+3*y^2-z^2-t^2,z^2+t^2]);
> Reduction(S,3);
Scheme over GF(3) defined by
$.1^2 + $.2^2,
$.3^2 + $.4^2
```

122.15 Maps between Schemes

Given schemes X and Y one can define a map $f\colon X \to Y$ 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 $f\colon X \to Y$ 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. The intrinsic `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 does not 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 $f(L)\colon X(L) \to Y(L)$. 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.

122.15.1 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 >: Sch, Sch, SeqEnum -> MapSch
map< X -> Y | F, G >: Sch, Sch, SeqEnum -> MapSch
map< X -> Y | u, F >: Sch, Sch, Map, SeqEnum -> MapSch
```

```
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 >: Sch, Sch, SeqEnum, SeqEnum -> MapAutSch`

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: Creating Maps of Affine Spaces (ex-fb50d9)

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);
```

(continues on next page)

(continued from previous page)

```
Scheme over Rational Field defined by
-x^2 + y^3 + 11*y^2 + 40*y + 48
```

Example: Maps Defined by Function Field Elements (ex-2e865b)

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: Frobenius Map (ex-3a4f79)

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): Sch -> MapSch

Create the identity map of the scheme X.

ConstantMap(X, Y, p): Sch, Sch, Pt -> MapSch

map< X -> Y | Q >: Sch, Sch, SeqEnum -> MapSch

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): Prj, Prj -> MapSch

The linear projection from projective space X to projective space Y that omits the first $\dim X - \dim Y$ coordinates.

Projection(X, Q): Sch, Prj -> Sch, MapSch

`Projection(X): Sch -> Sch, MapSch`

`Projection(X, p): Sch, Pt -> Sch, MapSch`

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): Sch, Pt -> Sch, MapSch, Sch`

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): [FldFunFracSchElt], Sch -> MapSch`

`ProjectiveMap(L): [FldFunFracSchElt] -> MapSch`

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): FldFunFracSchElt, Sch -> MapSch`

`ProjectiveMap(f): FldFunFracSchElt -> MapSch`

A short form for `ProjectiveMap([f,1],X,Y)`.

Example: Projective Map on a Plane Quartic (ex-cfe770)

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): Sch, SeqEnum -> Sch`

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): MapSch -> MapSch`

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): MapSch -> Bool, MapSch`

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): MapSch -> Bool`

Returns `true` if the map f has an inverse stored.

Example: Inverting a Map to an Elliptic Curve (ex-1608e5)

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: MapSch, MapSch -> MapSch`

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): Map -> [Map]`

The maps composed to form f.

Example: Hom Spaces of Schemes (ex-67bb10)

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 *Automorphisms*. 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 $f\colon B \to F$.

```
> f := map< B -> F | [x,y,z,t] >;
> Expand(ii(f)) eq Expand(i*f);
true
```

`Restriction(f, X, Y): MapSch, Sch, Sch -> MapSch`

```
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): MapSch -> MapSch`

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): MapSch -> MapSch`

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): MapSch -> MapSch

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): MapSch -> MapSch

Normalisation(phi): MapSch -> MapSch

The map created from ϕ by removing common factors from the defining polynomial.

Example: Converting a Map to a Morphism (ex-7d6748)

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
```

(continues on next page)

(continued from previous page)

```
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
```

`ImproveParametrization(p): MapSch -> MapSch`

Let p be a map over the rational numbers from the projective line, $\mathbf{P}^1_{\mathbb{Q}}$ =

`ProjectiveSpace(Rationals(),1)`, to another ordinary projective scheme X. For example, p could be a parametrization of a rational curve.

The intrinsic applies minimization and reduction for binary forms to construct and return a new map $\phi : \mathbf{P}^1 \to X$ over $\mathbb{Q}$ that is as simple as possible amongst those differing from p by an automorphism ψ (over $\mathbb{Q}$) of $\mathbf{P}^1$. That is, $\phi = p \circ \psi$ for some ψ.

Example: Improving a Conic Parametrization (ex-454322)

We start with a "bad" parametrization map of a plane conic, produced by choosing a large base point on the conic, and find a simpler parametrization using `ImproveParametrization`.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Conic(P2, y^2 - x*z - y*z);
> pt := C![1140, 2109, 1369];
> prm := Parametrization(C,pt);
> P1<s,t> := Domain(prm);
> prm;
 Mapping from: Curve over Rational Field defined by
0 to CrvCon: C
with equations :
1140/1369*s^2 + 77/37*s*t + t^2
57/37*s^2 + s*t
s^2
and inverse
z
y - 57/37*z
> prm1 := ImproveParametrization(prm);
> prm1;
Mapping from: Curve over Rational Field defined by
0 to CrvCon: C
with equations :
s*t
s^2 - s*t
s^2 - 2*s*t + t^2
and inverse
-y
-y + z
```

122.15.2 Basic Attributes

Trivial Attributes

`Domain(f): MapSch -> Sch`

The domain of the map of schemes f.

`Codomain(f): MapSch -> Sch`

The codomain of the map of schemes f.

`DefiningPolynomials(f): MapSch -> SeqEnum`

`DefiningEquations(f): MapSch -> SeqEnum`

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): MapSch -> SeqEnum`

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): MapSch -> SeqEnum`

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): MapSch -> SeqEnum`

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): MapSch -> SeqEnum`

The polynomials of all definitions of the map of schemes f.

`AllInverseDefiningPolynomials(f): MapSch -> SeqEnum`

The polynomials of all definitions of the inverse of the map of schemes f if f has a known inverse.

`AlgebraMap(f): MapSch -> Map`

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): MapSch -> RngIntElt`

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.

Basic Tests

`f eq g: MapSch, MapSch -> BoolElt`

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): MapSch -> BoolElt`

`IsPolynomial(f): MapSch -> BoolElt`

Returns `true` if and only if the map of schemes f is defined at all points of its domain.

`IsIsomorphism(f): MapSch -> BoolElt, IsoSch`

Returns `true` if and only if the map of schemes $f\colon X \to Y$ 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 $g\colon X \to Y$ which is of the recognised isomorphism type as a second value.

`IsDominant(f): MapSch -> BoolElt`

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): MapSch -> BoolElt`

Returns `true` if and only if the map of schemes f is a regular map defined by linear polynomials.

`IsAffineLinear(f): MapSch -> BoolElt`

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.

122.15.3 Maps and Points

Given a map $f\colon X \to Y$ 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 $f(K)\colon X(K) \to Y(K)$. 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): MapSch, Pt -> Pt`

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): MapSch, Pt -> Any`

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: Pt, MapSch -> Any`

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): MapSch, Rng -> Map`

`f(m): MapSch, Map -> Map`

The map induced by the map of schemes $f\colon X \to Y$ 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: Image of a Point (ex-59eef0)

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)
```

122.15.4 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): MapSch, Sch -> Sch`

`X @@ f: Sch, MapSch -> Sch`

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): MapSch -> Sch`

`f(X): Sch, MapSch -> Sch`

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 multi-graded, then X is saturated before the computation to ensure the correct result. For computational efficiency, we do not check that $X \cap U$ is indeed Zariski-dense in X.

`Image(f, X, d): MapSch, Sch, RngIntElt -> []`

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: Rational Quartic Curve as an Image `(ex-ba77c6)`

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);
```

(continues on next page)

(continued from previous page)

```
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: Image Equations of a Canonical Curve (ex-1cc82b)

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 *Algebraic Curves* 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): MapSch -> Sch`

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): MapSch -> SetEnum

BasePoints(f, L): MapSch, Fld -> SetEnum

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: Base Points of a Map (ex-40293b)

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: Elementary Transformation of Scrolls (ex-0c62af)

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.

122.15.5 Maps and Closure

`ProjectiveClosure(f): MapSch -> MapSch`

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): Aff, Prj, SeqEnum`

`MakePCMap(A, P, S): Aff, Prj, SeqEnum`

`MakeProjectiveClosureMap(m): Map`

`MakePCMap(m): Map`

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): MapSch, RngIntElt -> MapSch`

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): MapSch, RngIntElt, RngIntElt -> MapSch`

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: Restricting Maps to Patches `(ex-a968b3)`

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.

122.15.6 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): Sch, SeqEnum -> MapAutSch`

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 *Hyperelliptic Involution as an Automorphism*.

`IdentityAutomorphism(X): Sch -> MapAutSch`

`IdentityMap(X): Sch -> MapAutSch`

The identity map $X \to X$ on the scheme X.

`IsEndomorphism(f): MapSch -> BoolElt`

Returns `true` if and only if the domain and range of the maps of schemes f are equal.

`IsAutomorphism(f): MapSch -> BoolElt, AutSch`

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: Hyperelliptic Involution as an Automorphism (ex-e53b92)

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: Testing the Jacobian Conjecture (ex-b48ad5)

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
```

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): Sch, SeqEnum -> MapSch`

The automorphism of the affine space A determined by the sequence of functions F defined on A.

`Automorphism(A, M): Sch, Mtrx -> MapIsoSch`

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): Sch, Pt -> MapSch`

The translation map of the affine space A taking the rational point p to the origin.

`PermutationAutomorphism(A, g): Sch, GrpPermElt -> MapIsoSch`

`Automorphism(A, g): Sch, GrpPermElt -> MapIsoSch`

The automorphism of the affine space A that permutes its coordinates according to the permutation g.

Example: Permutation Automorphisms of Affine Space (ex-ea8d99)

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): Sch, RngMPolElt -> IsoSch`

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): MapSch -> MapSch, MapSch`

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: Decomposing an Affine Linear Map (ex-2c2ab5)

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): Aff -> MapSch

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): Aff, Mtrx -> MapAutSch

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: Projectivities of Affine Space (ex-c26942)

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;
```

(continues on next page)

(continued from previous page)

```
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
```

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): Prj, SeqEnum -> MapSch`

The automorphism of the projective space P determined by the sequence of polynomials F defined on P.

`Matrix(f): MapSch -> Mtrx`

The matrix corresponding to the linear automorphism f of a projective space.

`Automorphism(P, M): Sch, Mtrx -> MapSch`

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): Prj -> PowAutSch`

The parent of automorphisms of the projective space P.

`AutomorphismGroup(P): Prj -> GrpMat, Map`

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: Automorphisms of Projective Space as Matrices `(ex-a5702c)`

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): Prj, [Pt] -> MapSch`

The unique automorphism of the n-dimensional projective space P taking the $n+2$ standard simplex points $(1:0\cdots:0),\ldots$ and $(1:1\cdots:1)$ to the points of the sequence Q. The sequence Q must comprise $n+2$ linearly independent rational points of P.

`Translation(P, Q): Prj, [Pt] -> MapSch`

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 $(1:1\cdots:1)$ 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): Prj, Pt, Pt -> MapSch`

A choice of linear automorphism of the projective space P which takes the rational point p to the rational point q.

`Translation(X, p): Sch, Pt -> MapSch`

A choice of linear automorphism of the projective space P taking the point $(0:\cdots:0:1)$ to the rational point p.

Example: Translations of Schemes (ex-afaf79)

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): Prj -> MapSch`

`QuadraticTransformation(P, Q): Prj, SeqEnum -> MapSch`

The first function is the standard quadratic transformation of projective space P taking its coordinates to their reciprocals, that is $(x : y : \dots) \mapsto (1/x : 1/y : \dots)$. 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): Sch -> Sch, MapIsoSch`

`QuadraticTransformation(X, Q): Sch, [Pt] -> Sch, MapIsoSch`

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 trans-

formation 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: Factorising a Cremona Transformation (ex-3caa1e)

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
```

122.15.7 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 `MapSch`s in a number of contexts.

1 A graph map 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 bihomogeneous 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. For the image $f(S)$ of a subscheme S of X, the same restriction applies to S.

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): Sch, Sch, RngMPol -> MapSchGrph`

Saturated: BoolElt	Default: false

The schemes X and Y are ordinary projective schemes with ambients $\mathbf{P}^m$ and $\mathbf{P}^n$ respectively. The ideal I is in an $n+m+2$ variable polynomial ring R that should have the grevlex ordering. The ideal I should define the closure of the graph of a rational map from X to Y if R is identified 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.

Thus, the ideal I should 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

$$y_i F_j(x) - y_j F_i(x) \quad \forall\ 0 \leq i, j \leq n$$

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): MapSchGrph -> MapSch`

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 definingpolynomials are also added to the result.

`IsInvertible(f): MapSchGrph -> BoolElt, MapSchGrph`

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 `MapSch`s, which returns whether the map has already been determined to be invertible, is also available.

Example: Graph Map of an Elliptic Curve `(ex-19aa45)`

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
```

(continues on next page)

(continued from previous page)

```
> 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) }
```

122.16 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.

122.16.1 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): Sch -> Sch

```
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*). This will give the correct result as long as *no component of X lies in the hyperplane complement of the patch.*

IsInTangentVariety(X, P): Sch, Pt -> BoolElt

The computation of the full tangent variety can be quite time consuming except in small dimen-

sional 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: Tangent Variety (ex-239592)

```
> 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
```

122.16.2 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): Sch -> Sch

```
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): Sch, Pt -> BoolElt`

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: Secant Variety (ex-82e1f9)

```
> 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
```

122.16.3 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 practice 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): Sch -> Sch, MapSch`

```
IsoToSub: Verbose                    Default: 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): Crv -> Sch, MapSch

```
EmbCrv: Verbose                    Default: 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: Embedding a Plane Curve in Projective 3-Space (ex-f0b1d5)

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
```

122.17 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

$$a_1 f_1 + \cdots + a_r f_r = 0$$

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 does not change the hypersurface it defines and the equation $0 = 0$ does not 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 does not 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.

122.17.1 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.

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): Sch, RngIntElt -> LinearSys`

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): Sch, [RngIntElt] -> LinearSys`

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): Sch, SeqEnum[RngMPolElt] -> LinearSys`

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): Sch, [RngIntElt] -> SetIndx`

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: Equality of Linear Systems (ex-7aa596)

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): MapSch, Sch, RngIntElt -> LinearSys`

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: Finding Images with Linear Systems (ex-2e0ca6)

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].)

Geometric Restrictions: Points

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 determine a linear subsystem of a given linear system by imposing conditions on the sections of that system.

`LinearSystem(L, p): LinearSys, Point -> LinearSys`

Given a point p, construct the subsystem of the linear system L comprising of those hypersurfaces of L which include the point p.

`LinearSystem(L, P): LinearSys, SeqEnum[Point] -> LinearSys`

Given a sequence P of points, create the subsystem of the linear system L comprising of those hypersurfaces of L which include the points q of P.

Example: Subsystems Imposed at Points (ex-934794)

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 *Algebraic Curves* 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);
```

(continues on next page)

(continued from previous page)

```
> 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 do not 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
```

(continues on next page)

(continued from previous page)

```
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, p, m): LinearSys, Point, RngIntElt -> LinearSys`

`LinearSystem(L, P, M): LinearSys, SeqEnum[Point], SeqEnum[RngIntElt] -> LinearSys`

The arguments m and M give the multiplicities of the point p or the sequence of points P, repectively. If there is a sequence of points P then the sequence M contains the multiplicities of the points P where the i-th element of M is the multiplicity of the i-th point in P. The subsystem of L constructed in this case will consist of the hypersurfaces of L which pass through each point q of P with multiplicity at least n of M, where n is the corresponding element of M.

Example: Degree 20 Curve with 18 Singular Points (ex-9b3b7a)

In this example we construct a plane curve of degree 20 with 18 singular points having multiplicities $2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 5, 5, 5, 7, 7, 8, 9$.

```
> K := Rationals();
> A := AffineSpace(K, 2);
> M := [2,2,2,2,2,2,3,3,3,3,3,5,5,5,7,7,8,9];
> P := [A![Random(1,40), Random(1, 40)] : i in [1..#M]];
> time L := LinearSystem(LinearSystem(A, 20), P, M);
Time: 0.580
> #Sections(L);
1
> C := Curve(A, &+Sections(L));
> [Multiplicity(C, x): x in P];
[ 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 5, 5, 5, 7, 7, 8, 9 ]
```

Example: Subsystems through Thousands of Points (ex-114b8f)

The construction of a subsystem defined by points is very fast. We illustrate this with an example where the subsystem contains thousands of specified points.

```
> K := FiniteField(397);
> P := ProjectiveSpace(K, 3);
> pts := [P!([Random(K) : i in [1..3]] cat [1]):j in [1..3275]];
> L := LinearSystem(P, 25);
> #Sections(L);
3276
> #pts;
3275
> time J := LinearSystem(L, pts);
Time: 11.430
> #Sections(J);
1
```

Geometric Restrictions: Schemes

This restriction constructs the subsystem of a linear system L which consists of the elements of L which contain the scheme X.

`LinearSystem(L, X : parameters): LinearSys, Sch -> LinearSys`

The sections of this linear system are equal to the polynomials of the defining ideal of X whose homogeneous degree is the same as that of L.

If X is an affine scheme, then a Groebner basis for the ideal of X must be computed. If the parameter Order is set to "order", then the Groebner basis is computed using this order. The default is "grevlex". If X is projective, then the ideal of X must be saturated first, which takes time. This saturation can be avoided by setting the parameter `SaturateIdeal := false` (but in this case the result may not be correct).

Example: Unique Quadric Containing a Curve (ex-96c94f)

We define a curve in affine 3-space and show that there is exactly one quadric containing it.

```
> K := Rationals();
> A<x,y,z> := AffineSpace(K, 3);
> X := Curve(A, [x*y-y*z^2, x*y*z+y-1]);
> L := LinearSystem(A, 2);
> J := LinearSystem(L, X);
> J;

Linear system on Affine Space of dimension 3 over Rational Field
Variables: x, y, z
with 1 section:

x - z^2
```

Example: Subsystem from an Incomplete Linear System (ex-aeb538)

In this example the scheme X is a curve in 4-dimensional projective space and the initial linear system L is not complete.

```
> K := Rationals();
> R := PolynomialRing(K, 5, "grevlex");
> P := ProjectiveSpace(R);
> d := 5;
> s := [ &+[R.i^(d+j) : i in [1..5]]:j in [1..3] ];
> X := Scheme(P, s : Saturated := true);
> s := Sections(LinearSystem(P, d+4));
> s := [ &+[Random(1,10)*q : q in s] : i in [1 .. #s-30]];
> time L := LinearSystem(P, s);
> time J := LinearSystem(L, X);
Time: 4.220
> J;

Linear system with 25 sections

> // If L is complete, the computation is faster.

> L := LinearSystem(P, d+4);
> time J := LinearSystem(L,X);
Time: 0.000
> J;

Linear system with 55 sections
```

Example: Subsystem on a Product of Weighted Spaces (ex-4615cc)

In this example, the scheme X is a variety in a product of weighted projective spaces.

```
> K := FiniteField(103);
> P1 := ProjectiveSpace(K, [1, 2, 3, 5]);
> P2 := ProjectiveSpace(K, [2, 3, 5, 7]);
> P := DirectProduct(P1, P2);
> H := LinearSystem(P, [33, 35]);
> X := Scheme(P, [Random(H) : i in [1..3]] : Saturated:=true);
> L := LinearSystem(P, [35, 35]);
> time J := LinearSystem(L, X);
Time: 15.860
> #Sections(J);
6
```

Example: Affine and Projective Subsystem Timings (ex-8af960)

In this example, the time taken by a computation in affine space is compared with the time taken in the projective space.

```
> K := Rationals();
> R<x,y,z,w,t> := PolynomialRing(K, 5, "grevlex");
> A := AffineSpace(R);
> d := 9;
> s := [&+[R.i^(d+j) : i in [1..5]]:j in [1..3]];
> X := Scheme(A,[s[1], s[2]]) join Scheme(A, [s[2], s[3]]);
> L := LinearSystem(A, d+2);
> time J := LinearSystem(L, X);
Time: 10.550
> #Sections(J);
1
```

We now take the projective closure of the schene X before computing the linear system.

```
> K := Rationals();
> R<x,y,z,w,t> := PolynomialRing(K, 5, "grevlex");
> A := AffineSpace(R);
> d := 9;
> s := [&+[R.i^(d+j) : i in [1..5]] : j in [1..3]];
> X := Scheme(A, [s[1], s[2]]) join Scheme(A, [s[2], s[3]]);
> X := ProjectiveClosure(X);
> A := Ambient(X);
> L := LinearSystem(A, d+2);
> time J := LinearSystem(L, X);
Time: 0.010
> #Sections(J);
1
```

The projective computation is much faster as no Groebner basis computation is needed.

Geometric Restrictions: Affine Plane Curves with Non-ordinary Singularities

Let $S_1 \to S_0$ be the blow-up of a surface S_0 at a point p_0. Then S_1 contains an exceptional curve E_1 (isomorphic to the projective line) that is contracted to p_0. Let p_1 be a point in E_1. We can blow-up again at p_1 and choose a point p_2 in the new exceptional curve E_2. Iterating this we get a sequence of infinitely near points $(p_0, \ldots, p_n)$.

Assuming that S_0 is the affine plane, then p_0 is defined by coordinates $[a_0, b_0]$ in the plane, and p_i is defined by coordinates $[a_i : b_i]$ in the projective line E_i, for $i > 0$.

The coordinates $[a_i : b_i]$ have the geometric interpretation of tangent directions (direction of a line tangent to the branches of a curve singularity). After each blow-up, the new surface S_i is covered by affine plane patches, so we choose the one that contains the point p_i. At each step we choose coordinates

such that the new exceptional curve is always the line "$y = 0$". More precisely, the blow-up of a curve "$F(x, y) = 0$" is given by taking $F(x * y, y) = 0$, except if the tangent direction of the singularity is $[1 : 0]$, in which case we take $F(x, x * y) = 0$ followed by $(x, y) \to (y, x)$.

`LinearSystem(L, p, m, t): LinearSys, Point, SeqEnum, SeqEnum[SeqEnum] -> LinearSys`

Here L is a linear system of affine plane curves, p is a point in the plane, $m = [m_0, \ldots, m_n]$ is a sequence of non-negative integers and $t = [t_1, \ldots, t_n]$ is a sequence of vector coordinates (tangent directions). This function computes the subsystem of L containing the curves that have multiplicity $m_0, \ldots, m_n$ at the infinitely near points given by $p_0, t_1, \ldots, t_n$, respectively.

`LinearSystem(L, P, M, T): LinearSys, SeqEnum[Point], SeqEnum[SeqEnum], SeqEnum[SeqEnum[SeqEnum]] -> LinearSys`

Here L is a linear system of affine plane curves, P is a sequence of points in the plane, M is a sequence of sequences of non-negative integers and T is a sequence of sequences of vector coordinates. The sequence M gives the multiplicities at the infinitely near points given by $p, p - t_1, \ldots, p - t_n$, respectively where p runs though the points in P. This function allows the computation of plane curves with singularities of any type.

Example: Quartic Curve with a Tacnode (ex-766e2f)

A tacnode is a double point with two branches sharing the same tangent direction. Thus it is a double point which resolves to a double point after one blow-up. We compute a quartic curve with one tacnode. Then we use `ResolutionGraph` to check the result.

```
> K := Rationals();
> A<x,y> := AffineSpace(K, 2);
> p := A![0,0];
> m := [2,2];
> t := [[1,1]];
> J := LinearSystem(A, 4);
> L := LinearSystem(J, p, m, t);
> L;
Linear system on Affine Space of dimension 2 over Rational Field
Variables: x, y
with 9 sections:
x^4
x^3*y
x^3 - y^3
x^2*y^2
x^2*y - y^3
x^2 - 2*x*y + y^2
x*y^3
x*y^2 - y^3
y^4

> C := Curve(A, &+Sections(L));
> C;
x^4 + x^3*y + x^3 + x^2*y^2 + x^2*y + x^2 + x*y^3 + x*y^2 - 2*x*y + y^4 - 3*y^3
    + y^2
> ResolutionGraph(C, A![0,0]);

The resolution graph on the Digraph
Vertex  Neighbours

1 ([ -2, 2, 1, 0 ])     2 ;
2 ([ -1, 4, 2, 2 ])     ;
```

Example: Quadrifolium (ex-63a6fd)

Now let us compute the so-called quadrifolium: a curve of degree 6 with a quadruple point that

resolves to two different double points after one blow-up. Thus it looks like the union of two tacnodes with different tangent directions. We consider this as two different singularities of type $[4, 2]$ at the same point. In order to get a nicer picture, we consider curves symmetric with respect to the coordinate axes. We also ask that the curve contains, with multiplicity 1, three additional points, one of these with tangent directions $[[1, 1]]$ (we are fixing the tangent line at the point). When we do not impose a tangent direction, the corresponding sequence of tangent directions is the empty one: $[]$.

```
> K := Rationals();
> A<x,y> := AffineSpace(K, 2);
> s6 := Sections(LinearSystem(A, 6));
> s := [q : q in s6 | q eq Evaluate(q, [-x,y]) and q eq Evaluate(q, [x,-y])];
> J := LinearSystem(A, s);
> p := [A![0,0], A![0,0], A![1,1], A![2/10,7/10], A![7/10,2/10]];
> m := [[4,2], [4,2], [1,1], [1], [1]];
> t := [[[1,0]], [[0,1]], [[1,-1]], [], []];
> L := LinearSystem(J, p, m, t);
> f := Sections(L)[1];
> C := Curve(A, f);
> ResolutionGraph(C, A![0,0]);

The resolution graph on the Digraph
Vertex  Neighbours

1 ([ -2, 2, 1, 0 ])     2 ;
2 ([ -1, 4, 2, 2 ])     ;
```

Example: Curve with a Cusp Singularity (ex-faf2b2)

A cusp singularity is a double point which resolves, after one blow-up, to a smooth curve tangent to the exceptional curve. So in order to get a cusp we need to have a double point and then, after one blow-up, ask for multiplicity $[1, 1]$ with tangent direction $[1, 0]$. The linear system of quintic curves is of dimension 20. A cusp singularity imposes five conditions, so if we ask for four cusps, we expect to find exactly one such curve:

```
> K := Rationals();
> A<x,y> := AffineSpace(K,2);
> J := LinearSystem(A,5);
> p := [A![1,1],A![-1,1],A![1,-1],A![-1,-1]];
> m := [[2,1,1],[2,1,1],[2,1,1],[2,1,1]];
> t := [[[1,2],[1,0]],[[1,3],[1,0]],[[1,4],[1,0]],[[1,5],[1,0]]];
> L := LinearSystem(J,p,m,t);
> f := Sections(L)[1];
> f;

x^5 - 368/195*x^4*y + 77/195*x^4 + 14/13*x^3*y^2 - 40/13*x^3 - 112/195*x^2*y^3 -
    2/195*x^2*y^2 + 848/195*x^2*y - 152/195*x^2 + 1/13*x*y^4 - 16/13*x*y^2 +
    28/13*x - 32/195*y^5 - 7/195*y^4 + 176/195*y^3 + 16/195*y^2 - 512/195*y +
    68/195

> C := Curve(A,f);
> {ResolutionGraph(C, q) : q in p};

    The resolution graph on the Digraph
    Vertex      Neighbours

    1 ([ -3, 2, 1, 0 ]) 2 ;
    2 ([ -1, 6, 4, 1 ]) 3 ;
```

(continues on next page)

(continued from previous page)

```
    3 ([ -2, 3, 2, 0 ]) ;
}
```

Example: Two Cubics Meeting with Multiplicity 8 (ex-89e6c4)

We construct two cubics meeting at one point with multiplicty 8.

```
> K := Rationals();
> A<x,y> := AffineSpace(K, 2);
> J := LinearSystem(A, 3);
> p := A![0,0];
> m := [1,1,1,1,1,1,1,1];
> t := [[1,1], [1,2], [1,3], [1,4], [1,5], [1,6], [1,7]];
> L := LinearSystem(J, p, m, t);
> #Sections(L);
2
> C1 := Curve(A, Sections(L)[1]);
> C2 := Curve(A, Sections(L)[2]);
> Dimension(SingularSubscheme(C1));
-1
> Dimension(SingularSubscheme(C2));
-1
IntersectionNumber(C1, C2, p);
8
```

Example: Pencil of Curves with Tangency Conditions (ex-0bb25b)

Let C be a smooth plane conic not containing the origin p_0, and let $p_1, \ldots, p_6$ be points in C such that the lines through $p0, p1$ and p_0, p_2 are tangent to C. It can be shown the existence of a pencil L of curves of degree 8 with a quadruple point at p_0 and tacnodes at p_i with branches tangent to the line through p_0, p_i, for $i = 1, \ldots, 6$. Let f_1, f_2 be the defining polynomials of two such curves. The smooth resolution of the surface with equation $w^2 = f_1 f_2$ is a surface of general type with invariants $p_g = q = 1$ and $K^2 = 8$. Here we compute the pencil L.

```
> K := Rationals();
> A<x,y> := AffineSpace(K, 2);
> J := LinearSystem(A, 8);
> m := [[4], [2,2], [2,2], [2,2], [2,2], [2,2], [2,2]];
> p := [A![0,0], A![1,1], A![1,-1], A![5/2,2], A![5/2,-2], A![5,3], A![5,-3]];
> t := [   [],   [[1,1]], [[1,-1]], [[5/2,2]], [[5/2,-2]], [[5,3]], [[5,-3]]];
> L := LinearSystem(J, p, m, t);
> L;

Linear system on Affine Space of dimension 2 over Rational Field
Variables: x, y
with 2 sections:

x^8 - 2*x^7 - 17/2*x^6*y^2 + 18*x^5*y^2 + x^5 + 411/16*x^4*y^4 - 599/96*x^4*y^2 +
    1291/16*x^4 - 5245/96*x^3*y^4 - 15805/96*x^3*y^2 - 12275/384*x^2*y^6 +
    29075/192*x^2*y^4 - 36275/384*x^2*y^2 + 2125/96*x*y^6 + 14125/96*x*y^4 +
    6875/384*y^8 - 23125/192*y^6 + 6875/384*y^4
x^6 - x^5*y^2 - x^5 + 1/4*x^4*y^4 - 553/144*x^4*y^2 + 1/4*x^4 + 625/144*x^3*y^4 +
    625/144*x^3*y^2 - 625/576*x^2*y^6 + 625/288*x^2*y^4 - 625/576*x^2*y^2 -
    625/144*x*y^6 - 625/144*x*y^4 + 625/576*y^8 + 625/288*y^6 + 625/576*y^4

> C:=Curve(A, &+Sections(L));
> {ResolutionGraph(C, q) : q in p};
```

(continues on next page)

(continued from previous page)

```
{
    The resolution graph on the Digraph
    Vertex      Neighbours

    1 ([ -1, 4, 1, 4 ]) ;

    ,
    The resolution graph on the Digraph
    Vertex      Neighbours

    1 ([ -2, 2, 1, 0 ]) 2 ;
    2 ([ -1, 4, 2, 2 ]) ;

}
```

Geometric Restrictions: Trace on a Scheme

`LinearSystemTrace(L, X): LinearSys, Sch -> LinearSys`

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: Trace on a Twisted Cubic (ex-932066)

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 *Geometrical Properties*.)

```
> 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 *Linear Systems and Maps*.)

```
> 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
```

Explicit Restrictions

`LinearSystem(L, F): LinearSys, SeqEnum -> LinearSys`

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): LinearSys, ModTupFld -> LinearSys`

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.

122.17.2 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.

Tests for Linear Systems

`Ambient(L): LinearSys -> Prj`

`AmbientSpace(L): LinearSys -> Prj`

The projective space on which the linear system L is defined.

`L eq K: LinearSys, LinearSys -> BoolElt`

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): LinearSys -> BoolElt`

Returns `true` if and only if the linear system L is the complete linear system of polynomials of some degree.

`IsBasePointFree(L): LinearSys -> BoolElt`

`IsFree(L): LinearSys -> BoolElt`

Returns `true` if and only if the linear system L has no base points.

Geometrical Properties

`Sections(L): LinearSys -> SeqEnum`

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): LinearSys -> RngMPolElt`

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): LinearSys -> RngIntElt`

The degree of the sections of the linear system L.

`Dimension(L): LinearSys -> RngIntElt`

The projective dimension of the linear system L. This is the maximal number of linearly independent sections of L minus 1.

`BaseScheme(L): LinearSys -> SchProj`

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): LinearSys -> SchProj`

The hypersurface common to all the elements of the linear system L.

`Reduction(L): LinearSys -> LinearSys`

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: Reducibility from Too Many Conditions (ex-ad2e4b)

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
]
```

(continues on next page)

(continued from previous page)

```
> [ 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): LinearSys -> SeqEnum`

A sequence containing the basepoints of the linear system L if the base locus of L is finite dimensional.

`Multiplicity(L, p): LinearSys, Pt -> RngIntElt`

The generic multiplicity of hypersurfaces of the linear system L at the point p.

Linear Algebra

`CoefficientSpace(L): LinearSys -> ModTupFld`

The vector space corresponding to the linear system L whose vectors comprise the coefficients of the polynomial sections of L.

`CoefficientMap(L): LinearSys -> ModTupFldElt`

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): LinearSys -> RngMPolElt`

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): LinearSys, LinearSys -> LinearSys`

A maximal subsystem of the linear system L which does not contain any of the hypersurfaces of the linear system K.

`Complement(L, X): LinearSys, Sch -> LinearSys`

A maximal subsystem of the linear system L comprising hypersurfaces not containing X.

Example: Linear Systems from Subspaces (ex-0ff141)

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
```

(continues on next page)

(continued from previous page)

```
> 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: LinearSys, LinearSys -> LinearSys

Intersection(L, K): LinearSys, LinearSys -> LinearSys

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: Sch, LinearSys -> BoolElt

Returns **true** if and only if the scheme X occurs among the hypersurfaces comprising the linear system L.

f in L: RngMPolElt, LinearSys -> BoolElt

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: LinearSys, LinearSys -> BoolElt

IsSubsystem(L, K): LinearSys, LinearSys -> BoolElt

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.

122.17.3 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 *Maps between Schemes*. 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): MapSch, LinearSys -> LinearSys

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.

122.18 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 can be we explicitly checked in general using the **IsCartier** intrinsic. There are also a number of intrinsics that are specific to surfaces.

The scheme X may be singular but is assumed to be a *normal* variety. In the absence of normality, problems may occur at along the non-normal locus L. In particular, any codimension 1 irreducible components of L will function badly as divisor components. Even when L is of codimension ≥ 2 in X, we may get strange behaviour at points of L. For example, a locally-principal (Cartier) effective divisor passing through a non-normal point p may become non-Cartier at p when its defining ideal I is replaced by the "pure codimension 1" component of I. This replacement is carried out as a matter of course in certain places where we need to kill off higher-codimensional components of the defining ideal of an effective divisor that would affect a computation.

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.

122.18.1 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): Sch -> DivSch`
: The divisor group of variety X.

`Variety(G): DivSch -> Sch`
: The variety of the divisor group G.

`G1 eq G2: DivSch, DivSch -> BoolElt`
: Returns `true` if and only if the divisor groups $G1$ and $G2$ are for the same variety.

122.18.2 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): Sch, FldFunFracSchElt -> DivSchElt`

`Divisor(X, f): Sch, FldFunRatMElt -> DivSchElt`

`Divisor(X, f): Sch, RngMPolElt -> DivSchElt`
: 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): Sch, SeqEnum -> DivSchElt`

`Divisor(X, Y): Sch, Sch -> DivSchElt`

`Divisor(X, I): Sch, RngMPol -> DivSchElt`

```
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.

The parameter `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. The parameter `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): Sch -> DivSchElt`

Creates a divisor given by a hyperplane section of projective variety X.

`ZeroDivisor(X): Sch -> DivSchElt`

The zero divisor on variety X.

`CanonicalDivisor(X): Sch -> DivSchElt`

A canonical divisor on variety X which 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): ShfCoh -> DivSchElt`

The argument S is a coherent sheaf that should be invertible (locally free, rank 1) on a variety X. Then S is 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): DivSchElt -> DivSchElt`

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): DivSchElt -> DivSchElt`

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): DivSchElt -> DivSchElt`

Returns $D-$ `RoundDownDivisor`(D).

`IntegralMultiple(D): DivSchElt -> DivSchElt, RngIntElt`

Finds a positive integer N such that $E = N * D$ is an integral divisor. Returns E and N. This does not attempt to find the smallest possible N by analysing the full prime factorisation of D.

`EffectiveHypersurfaceTwist(D): DivSchElt -> DivSchElt, RngMPolElt`

The divisor D should be an integral divisor on an ordinary projective variety X. Let H denotes a hyperplane divisor of X. Returns an effective divisor D_1 linearly equivalent to $D + rH$ for some $r \geq 0$. In fact, it also returns a non-zero homogeneous polynomial F of degree r in the coordinate ring of the ambient of X such that $D_1 = D + (F)$ where $(F) \sim rH$ is the divisor of vanishing of F on X. The D_1 returned will have an effective ideal factorisation (i.e., all multiplicities m_i in the stored factorisation will be positive integers). If D already has an effective ideal factorisation, D itself is returned with $F = 1$.

122.18.3 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): DivSchElt -> RngMPol`

Returns the defining ideal for an effective, integral divisor D.

`Support(D): DivSchElt -> Sch`

The subscheme of the variety of effective $\mathbf{Q}$-divisor D that gives its support.

`IdealOfSupport(D): DivSchElt -> RngMPol`

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): DivSchElt -> DivSchElt, DivSchElt`

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): DivSchElt -> SeqEnum`

Returns the current stored factorisation of D as a sequence of pairs of ideals and rational multiplicities.

`CombineIdealFactorisation(~D): DivSchElt`

Simplify the current ideal factorisation of D by combining terms with the same ideal.

`ComputeReducedFactorisation(~D): DivSchElt`

`ReducedFactorisation(D): DivSchElt -> SeqEnum`

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): DivSchElt`

`PrimeFactorisation(D): DivSchElt -> SeqEnum`

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): DivSchElt, DivSchElt -> FldRatElt`

The multiplicity of prime divisor E in divisor D. The ideal factorisation of D is replaced by a factor representing E to the multiplicity along with factors which are the original ideal factors of D with the maximum powers of E divided out. This refactorisation comes from the computation which iteratively calculates the colon ideal by E of the original factors.

`MultiplicityFast(D, E): DivSchElt, DivSchElt -> FldRatElt`

This is a fast version of `Multiplicity` that can be applied when the variety X of D and E is affine or ordinary projective and E is known to be Cartier. It is usually faster than the general version, not having to compute colon ideals, but has the drawback of not refactorising the ideal factorisation of D into a power of E and remainders. As a fast version, it also does not check that E is prime whereas the usual multiplicity intrinsic does.

`Multiplicities(D, P): DivSchElt, SeqEnum[DivSchElt] -> SeqEnum`

This is a fast (usually) variant of `Multiplicity` for working with divisors that are all supported on a known set of prime divisors. The argument D is a divisor on variety X and P should be a sequence of prime divisors containing *all* of the primes that D is supported on. Returns the sequence of multiplicities of p in D for $p \in P$. X must be ordinary projective here.

The method used to is reduce to the dimension 1 case for X by cutting with random hyperplanes and then use Magma's divisor and place functionality for curves to compute the result. We have to be careful in choosing hyperplanes such that at each stage of reducing the dimension by 1: the cut X is purely of dimension one less than before; all of the cut divisors p are purely of dimension one less than before and have no component lying in the singular locus of cut X; no components of a pair of cut p coincide.

The reason for requiring that all prime factors in the support of D lie in P is that other factors of D outside of P could end up coinciding with a factor p in P on repeated cutting by hyperplanes. This would lead to a multiplicity for p that is too large. This actually occurs in the example below.

This intrinsic is particularly useful when X lies in a fairly small dimensional ambient and the user is interested in computing multiplicities of a number of divisors that are supported on a fixed set of prime divisors and can have quite high multiplicities with respect to these.

Example: Divisor Multiplicities on a Cubic Surface (ex-ce56dc)

We take a (singular but normal) degree 3 Del Pezzo surface S in $\mathbf{P}^3$ and 3 lines on it that form a hyperplane section H as our initial set of P. We let D be the hypersurface divisor with equation the cube of the equation of H multiplied by the square of a different linear form that vanishes on the third line but not the other two. The expected multiplicities are thus $[3, 3, 5]$. However, not including the second prime divisor of the second linear form leads to the wrong result! The chosen cutting merges the ignored prime divisor of the support with the third line and leads to a value for the third multiplicity that is too high. When we include the final divisor of the support of D in our set P, we get the correct result.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> F := x^3+y^3+z^3+3*y^2*t+3*y*z*t+3*z^2*t+3*y*t^2+3*z*t^2+t^3;
> S := Surface(P,F);
> R := CoordinateRing(P);
> Ils := [ideal<R|[u,x+y+z+t-u]> : u in [x,y,z]]; //ideals of the three lines
> P := [Divisor(S,I) : I in Ils];
> g1 := x+y+z+t; // hyperplane = the sum of the lines
> g2 := x+y+t;   // hyperplane through only the 3rd line
> D := Divisor(S,g1^3*g2^2);
> Multiplicities(D,P);
[3,3,9]
> // wrong! we need to include the extra prime divisor of the support.
> D1 := Divisor(S,g2);
> PrimeFactorisation(D1);
[
    <Ideal of Polynomial ring of rank 4 over Rational Field
    Order: Graded Reverse Lexicographical
    Variables: x, y, z, t
    Homogeneous, Dimension 2, Radical, Prime
    Groebner basis:
    [
        z^2 + 3*y*t + 3*z*t + 3*t^2,
        x + y + t
    ], 1>,
    <Ideal of Polynomial ring of rank 4 over Rational Field
    Order: Graded Reverse Lexicographical
    Variables: x, y, z, t
    Homogeneous, Dimension 2, Radical, Prime
    Groebner basis:
    [
        x + y + t,
        z
    ], 1>
]
> P4 := Divisor(S,$1[1][1]);
> Multiplicities(D,P cat [P4]); //this gives the correct result!
[ 3, 3, 5, 2 ]
```

122.18.4 Basic Divisor Predicates

IsZeroDivisor(D): DivSchElt -> BoolElt

Returns whether D is the zero divisor.

IsIntegral(D): DivSchElt -> BoolElt

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): DivSchElt -> BoolElt`

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): DivSchElt -> BoolElt`

Returns whether D is a prime divisor.

`IsFactorisationPrime(D): DivSchElt -> BoolElt`

Returns whether the current factorisation of D is a prime factorisation (i.e. all ideals occurring are prime).

`IsDivisible(D): DivSchElt -> BoolElt, RngIntElt`

Returns whether D is integral and divisible as an integral divisor by an integer $n > 1$. If so, also returns the maximum such n.

122.18.5 Arithmetic of Divisors

`D1 + D2: DivSchElt, DivSchElt -> DivSchElt`

`D1 + D2: DivSchElt, DivTorElt -> DivSchElt`

`D1 + D2: DivTorElt, DivTorElt -> DivSchElt`

`D1 - D2: DivSchElt, DivSchElt -> DivSchElt`

`D1 - D2: DivSchElt, DivTorElt -> DivSchElt`

`D1 - D2: DivTorElt, DivSchElt -> DivSchElt`

`- D: DivSchElt -> DivSchElt`

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: RngIntElt, DivSchElt -> DivSchElt`

`r * D: FldRatElt, DivSchElt -> DivSchElt`

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: DivSchElt, DivSchElt -> BoolElt`

Returns whether divisors $D1$ and $D2$ lie on the same variety and are equal.

122.18.6 Further Divisor Properties

More complicated predicates on divisors.

`IsCanonical(D): DivSchElt -> BoolElt`

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): DivSchElt -> BoolElt`

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): DivSchElt -> BoolElt, RngIntElt`

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): DivSchElt -> BoolElt, FldFunFracSchElt`

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.

`IsCartier(D): DivSchElt -> BoolElt`

Returns whether D is a Cartier divisor on its variety X. This is equivalent to D being *locally-principal*. That is, X can be covered by Zariski-open subsets U_i such that the restriction of D to each U_i is equal to the restriction of $div(f_i)$ to U_i for some f_i. X must be ordinary projective here.

Example: Cartier Divisors `(ex-8f4659)`

Many divisors are automatically Cartier. If X is non-singular, all divisors are. If X is ordinary projective, the easiest divisors to construct are the hypersurface divisors defined by a homogeneous form that doesn't vanish on all of X. These are also all Cartier whether X is singular or not.

On the other hand, on an anticanonically-embedded degenerate Del Pezzo surface, a line that passes through a singular point is usually not Cartier. We give an example of a degenerate degree 4 Del Pezzo with 2 A_1 singularities where we verify this for a particular line using `IsCartier`.

```
> P<x,y,z,t,u> := ProjectiveSpace(RationalField(), 4);
> S := Surface(P,[z^2-z*t-y*u, y*z-x*t-z*t-y*u]);
> p := S![0,0,0,0,1];
> IsSingular(p);
true (0 : 0 : 0 : 0 : 1)
> l := Scheme(P,[y,z-t,x+z]); // l is a line thorugh p
> l subset S; // and l lies in S
true
> D := Divisor(S,Ideal(l));
> IsCartier(D);
false
```

`IsLinearlyEquivalent(D, E): DivSchElt, DivSchElt -> BoolElt, FldFunFracSchElt`

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 $D = E + \mathrm{div}(f)$. Uses `IsPrincipal` for the difference between the two divisors, so X must be ordinary projective.

`BaseLocus(D): DivSchElt -> Sch`

`IsBasePointFree(D): DivSchElt -> BoolElt`

`IsMobile(D): DivSchElt -> BoolElt`

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): DivSchElt, DivSchElt -> FldRatElt`

The arguments $D1$ 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): DivSchElt -> FldRatElt`

The intersection number of D with itself where the divisor D on X must satisfy the conditions for the last intrinsic.

`Degree(D): DivSchElt -> FldRatElt`

`Degree(D, H): DivSchElt, DivSchElt -> FldRatElt`

The argument 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): DivSchElt -> BoolElt`

The argument 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): DivSchElt -> BoolElt`

The divisor D on X is as above. The divisor D must be effective. Returns whether D is a nef divisor *and* has positive self-intersection.

`NegativePrimeDivisors(D): DivSchElt -> SeqEnum`

The divisor D on 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): DivSchElt -> DivSchElt, DivSchElt`

The divisor D on 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).

`Reduction(D, p): DivSchElt, Any -> DivSchElt`

```
parent: Sch                    Default: false
```

The variety X of the divisor D should be an ordinary projective scheme defined over K, the rational field or a number field. If K is the rationals then argument p should be a rational prime and if K is a number field, p should be a place or a prime ideal in an order of K.

Computes and returns D_p, the reduction mod p of D, which is a divisor on X_p, the reduction of the flat closure of X as defined by the intrinsic (*Reduction*). Note that it is not checked whether X_p is normal or even a variety over k_p (the residue class field of p). The reduction is carried out on the components in the current ideal factorisation of D in the same way the reduction of X itself is computed.

If the reduction of X has already been computed as Y, say, and the user wishes the result D_p to be a divisor on Y, this can be achieved by setting the parameter `parent` equal to Y.

122.18.7 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): DivSchElt -> ShfCoh`

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): DivSchElt -> SeqEnum`

`RiemannRochSpace(D): DivSchElt -> ModTupFld, Map`

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. The variety 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): Any, DivSchElt -> BoolElt, SeqEnum`

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): DivSchElt -> BoolElt, DivSchElt`

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.

122.19 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): Sch -> Map`

`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): Sch, SeqEnum -> List`

```
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): Sch, SeqEnum -> BoolElt, Pt`

```
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): Sch, SeqEnum -> BoolElt, SeqEnum`

```
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: Large Integral Points on Elliptic Curves `(ex-f7d305)`

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:

$$X(t) = t^4 + t^3 + x_2t^2 + x_1t + x_0, Q(t) = t^2 + q_1t + q_,0$$

$$Y(t) = t^5 + y_3t^3 + y_2t^2 + y_1t + y_0, A(t) = a_0, B(t) = b_1t + b_0.$$

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: Solutions to Hall's Conjecture (ex-0b5832)

Another application of the isolated points routine 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: Isolated Points of a Random Scheme (ex-b50f35)

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];
1812403568722098960*x^22 + 6297705584492967832*x^21 +
 6527336365144235612 8*x^20 + 8127120407582645599 2*x^19 +
 1307013696001389696 80*x^18 - 2853763840612678416 22*x^17 -
 8021669561188154716 54*x^16 + 2533254447903279968 45*x^15 -
 1266591733002155213172*x^14 + 2511386184440323009 0*x^13 +
 5065309674068046314 82*x^12 - 1323179973699695447463*x^11 +
 1605685921502538803112*x^10 - 1318315736155520576802*x^9 +
 9496491295829584599 58*x^8 - 5274413325441713384 90*x^7 +
 2544636840498666075 12*x^6 - 1000391891985775814 40*x^5 +
 2601441129568647585 6*x^4 - 3177984195514332576*x^3 -
 185294668718029075 2*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 - 708694145182058395 37/142324397561167099 52*y^21 -
    606754258610022348837 3/569297590244668398 08*y^20 + [...]
    [...] + 166661449939161/1779054969514588744
> MinimalPolynomial(V[22][3]); // deg 22, all coeffs about 25 digits
y^22 - 428567519465749893/68067993818308256*y^21 -
    229592953968800596 15/108908790109293209 6*y^20 + [...]
    [...] + 246916540549044 1431/68067993818308256
> 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: Polynomial with Mathieu Monodromy (ex-20eb23)

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 $f = P_3P_2^2P_4^4 = P_7P_8^2 - c$ 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;
```

(continues on next page)

(continued from previous page)

```
> 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 ]
]
```

122.20 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.

122.20.1 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: Twisted Cubics `(ex-269de3)`

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;
```

(continues on next page)

(continued from previous page)

```
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 := SingularPointsOverAlgebraicClosure(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
```

(continues on next page)

(continued from previous page)

```
    (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.

122.20.2 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: Curves in Space `(ex-31b8fc)`

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.

122.21 Bibliography

- [AGZV85] V. I. Arnold, S. M. Gusein-Zade, and A. N. Varchenko. *Singularities of Differentiable Maps Volume I*. Volume 82 of Monographs in Mathematics. Birkhauser, 1985.
- [Bru] N. Bruin. "Some ternary Diophantine equations of signature $(n, n, 2)$."
- [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.

Chapter 123

Coherent Sheaves

123.1 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.

123.2 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): ModMPol, Sch -> ShfCoh

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): Sch -> ShfCoh

StructureSheaf(X, n): Sch, RngIntElt -> ShfCoh

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): Sch -> ShfCoh

CanonicalSheaf(X, n): Sch, RngIntElt -> ShfCoh

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): ShfCoh, RngIntElt -> ShfCoh

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: Sheaves on a Cubic Surface (ex-ac24b9)

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): Sch -> ShfCoh`

```
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): Sch -> ShfCoh`

```
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): Prj -> ShfCoh`

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.

123.3 Accessor Functions

The following functions provide an interface to conveniently extract the basic data from a coherent sheaf.

`Module(S): ShfCoh -> ModMPol`

Returns the graded module that was used to define sheaf S.

`Scheme(S): ShfCoh -> Sch`

Returns the ordinary projective scheme X on which the sheaf S is defined.

`FullModule(S): ShfCoh -> ModMPol`

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. That is, M has no embedded associated primes (except possibly the redundant ideal) 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., torsion-free sheaves 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): ShfCoh -> ModMPol`

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): ShfCoh`

Procedure to compute and store (but not return) the maximal module M_{max} of the sheaf S.

Example: Sheaf of an Unsaturated Module (ex-0b6d9f)

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
```

123.4 Basic Constructions

The following functions give some basic constructions on sheaves.

TensorProduct(S, T): ShfCoh, ShfCoh -> ShfCoh

```
Maximize: BoolElt                    Default: false
```

TensorPower(S, n): ShfCoh, RngIntElt -> ShfCoh

```
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): ShfCoh -> ShfCoh`

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): ShfCoh, ShfCoh -> ShfCoh, Map`

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): ShfCoh, ShfCoh -> ShfCoh`

For S and T sheaves on the same scheme X, this function returns the sheaf direct sum $S \oplus T$.

`Restriction(S, Y): ShfCoh, Sch -> ShfCoh`

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: Ruling on a Quadric Cone (ex-22d504)

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]
```

123.5 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): ShfCoh, ShfCoh, ModMPolHom -> ShfHom`

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): ShfHom -> ShfCoh`

The domain of the sheaf homomorphism f.

`Codomain(f): ShfHom -> ShfCoh`

The codomain of the sheaf homomorphism f.

`Degree(f): ShfHom -> RngIntElt`

The degree of the sheaf homomorphism f as defined in the introduction to this section.

`ModuleHomomorphism(f): ShfHom -> ModMPolHom`

The underlying homogeneous graded module homomorphism of the sheaf homomorphism f.

`Kernel(f): ShfHom -> ShfCoh, ShfHom`

Given a sheaf homomorphism f, this function returns the kernel of f and its inclusion homomorphism into the domain of f.

`Image(f): ShfHom -> ShfCoh, ShfHom, ShfHom`

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): ShfHom -> ShfCoh, ShfHom`

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): SeqEnum[ShfHom] -> ShfHom`

If $hms = [h_1, \ldots, h_n]$ is a sequence of sheaf homomorphisms, then this function returns the composition of homomorphisms $h_1 * h_2 * \ldots h_n$. 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.

123.6 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): ShfCoh -> Map, Sch`

`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 $X \to \mathrm{Proj(R)} \to \mathbf{P}^{\mathrm{r}}$ 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 *Scheme Graph Maps*). 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): Sch, RngMPol -> ShfCoh`

`RiemannRochBasis(X, I): Sch, RngMPol -> SeqEnum, RngMPolElt, ShfCoh`

`GetMax: BoolElt`	`Default: true`

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.

```
IneffectiveDivisorToSheaf(X, I, J): Sch, RngMPol, RngMPol -> ShfCoh
IneffectiveRiemannRochBasis(X, I, J): Sch, RngMPol, RngMPol -> SeqEnum, RngMPolElt,
                            ShfCoh
```

```
GetMax: BoolElt                    Default: true
```

These are extensions of the above 2 intrinsics that compute the sheaf or Riemann-Roch basis for a non-effective divisor that can be represented as a difference $D - E$ of two effective locally principal divisors D, E, which have no irreducible component in common. Instead of having a single ideal to represent the divisor, the divisor is represented by two ideals I and J which are the ideals of the coordinate ring of the ambient of variety X that define respectively the effective divisors D and E as subschemes of the ambient.

The algorithm is a fairly straightforward variant of the effective divisor version.

Example: Divisor Map of a Rational Scroll `(ex-72931f)`

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 $\mathbb{P}^1$.

```
> 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(Sl : 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: Blowing Down Lines by a Divisor Map (ex-774e58)

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
```

123.7 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): ShfCoh -> BoolElt, RngIntElt

```
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 practice.

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): ShfCoh, ShfCoh -> BoolElt, ShfHom`

`IsIsomorphicWithTwist(S, T): ShfCoh, ShfCoh -> BoolElt, RngIntElt, ShfHom`

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): ShfCoh -> BoolElt`

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.

123.8 Miscellaneous

`CohomologyDimension(S, r, n): ShfCoh, RngIntElt, RngIntElt -> RngIntElt`

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): ShfCoh -> RngIntElt`

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): ShfCoh, ShfCoh -> RngIntElt`

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): ShfCoh, ModMPolElt -> Sch`

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 $\langle f_1, \ldots, f_n \rangle$.

123.9 Examples

In this section we present some extended examples illustrating various features of the sheaf machinery.

Example: Rational Surface of Degree 10 `(ex-1d7a31)`

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] +
```

(continues on next page)

(continued from previous page)

```
>         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 +
```

(continues on next page)

(continued from previous page)

```
>         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 +
```

(continues on next page)

(continued from previous page)

```
>         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] +
```

(continues on next page)

(continued from previous page)

```
>         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 +
```

(continues on next page)

(continued from previous page)

```
>         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] +
```

(continues on next page)

(continued from previous page)

```
>         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: Normalising a Coordinate Ring (ex-362695)

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 $\mathbb{P}^3$ over $\mathbf{Q}$. The curve C can be thought of as having been embedded in $\mathbb{P}^7$ by a complete linear system of degree 8 and then (isomorphically) projected down into $\mathbb{P}^3$. Such genus one curves embedded as degree 8 curves in $\mathbb{P}^3$ 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 $\mathbb{P}^7$. The coordinate ring of this is isomorphic to the normalisation of the coordinate ring of C in $\mathbb{P}^3$. 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 $\mathbb{P}^3$, 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 +
```

(continues on next page)

(continued from previous page)

```
> 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 ]
```

123.10 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.

Chapter 124

Algebraic Curves

124.1 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 *Schemes*. 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.

124.1.1 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.

124.1.2 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.

124.1.3 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
```

124.1.4 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
```

124.1.5 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 Newton Polygons.

```
> 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.

124.1.6 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 *Schemes*. 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 Algebraic Function Fields). 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
```

124.2 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 *Schemes* for more constructors and functions.

AffineSpace(k, n): Rng, RngIntElt -> Aff

AffinePlane(k): Rng -> Aff

Create affine n-dimensional space (resp. the 2-dimensional affine plane) over the ring k.

ProjectiveSpace(k, n): Rng, RngIntElt -> Prj

ProjectivePlane(k): Rng -> Prj

Create n-dimensional projective space (resp. the 2-dimensional projective plane) over the ring k.

DirectProduct(A, B): Prj, Prj -> PrjProd, SeqEnum

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): Rng, RngIntElt -> PrjScrl

RuledSurface(k, a, b): Rng, RngIntElt, RngIntElt -> PrjScrl

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): Sch -> RngMPol

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): Aff -> FldFunFracSch`

`FunctionField(A): Prj -> FldFunFracSch`

Return the function field of the ambient space A of dimension 1. 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.

While it is possible to give input of dimension greater than 1 for backward compatibility, it is preferrable to instead use `FieldOfFractions(CoordinateRing(X))`.

`A ! [a,...]: Sch, [RngElt] -> Pt`

`A(L) ! [a,...]: SetPt, [RngElt] -> Pt`

For elements $a, \ldots$ 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 $(a, \ldots)$ where these coordinates are elements of L (or the base ring of A).

`Origin(A): Aff -> Pt`

The point $(0, 0, \ldots, 0)$ of the affine space A.

`Coordinates(p): Pt -> SeqEnum`

`p[i]: Pt, RngIntElt -> RngElt`

The complete sequence of base ring elements corresponding to the coordinates of the point p or the ith coordinate or p alone.

Example: Points of an Affine Space (ex-300d2e)

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
```

124.3 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:

$$C\colon (f_1 = f_2 = \ldots = f_n = 0) \subset A.$$

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.

124.3.1 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): Sch, RngMPolElt -> CrvPln`

```
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): Sch, RngMPol -> Crv`

```
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): Sch, SeqEnum -> Crv`

```
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): Sch -> BoolElt, Crv`

Returns `true` if and only if X is a one-dimensional scheme.

`Curve(X): Sch -> Crv`

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): CrvPln, Pt, Pt -> CrvPln`

`Line(P, S): Prj, {Pt} -> Sch`

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): Prj, {Pt} -> Crv`

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 Rational Curves and Conics for details of the special functions that apply in that case.

`Union(C, D): Sch, Sch -> Sch`

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.

124.3.2 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 $m\colon k \to L$ 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): Sch, Rng -> Sch`

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): Sch, Map -> Sch`

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): Sch, Sch -> Sch`

`BaseChange(C, A, m): Sch, Sch, Map -> Sch`

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): Sch, RngIntElt -> Sch`

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: Base Change of a Singular Curve (ex-d5a0e9)

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) }
```

124.3.3 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 *Function Fields*.

AmbientSpace(C): Sch -> Sch

The ambient space containing the curve C.

BaseRing(C): Sch -> Rng

CoefficientRing(C): Sch -> Rng

BaseField(C): Sch -> Fld

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): Sch -> RngMPolElt

The defining polynomial of the plane curve C.

DefiningIdeal(C): Sch -> RngMPol

The defining ideal of the curve C, as an ideal in the coordinate ring of its ambient space.

CoordinateRing(C): Sch -> Rng

The coordinate ring of the curve C. Even creating this requires the use of Gröbner basis techniques.

Degree(C): Sch -> RngIntElt

The degree of the of the curve C which must be defined in an ordinary projective ambient space.

`JacobianIdeal(C): Sch -> RngMPol`

The ideal of partial derivatives of the defining polynomials of the curve C.

`JacobianMatrix(C): Sch -> ModMatRngElt`

The matrix of partial derivatives of the defining polynomials of the curve C.

`HessianMatrix(C): Sch -> Mtrx`

The symmetric matrix of second partial derivatives of the defining polynomial of the plane curve C.

Example: Hessian of a Plane Curve (ex-2ecc15)

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) }
```

124.3.4 Basic Invariants

`IsReduced(C): Sch -> BoolElt`

Returns `true` if and only if the ideal defining the curve C is reduced.

`IsIrreducible(C): Sch -> BoolElt`

Returns `true` if and only if the curve C is irreducible (as a scheme).

`IsSingular(C): Sch -> BoolElt`

Returns `true` if and only if the curve C contains at least one singularity over an algebraic closure of its base field.

`IsNonsingular(C): Sch -> BoolElt`

Returns `true` if and only if the curve C has no singularities over an algebraic closure of its base field.

124.3.5 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): RngIntElt, RngIntElt, Prj -> CrvPln`

`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 $\mathbb{Q}$. Over $\mathbb{Q}$, 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): Crv -> BoolElt`

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): RngIntElt, SeqEnum, Prj -> CrvPln, RngMPol`

`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 $\mathbb{Q}$ and `RandomBound` is as before.

`RandomCurveByGenus(g, K): RngIntElt, Fld -> Crv`

`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 $\mathbb{Q}$. The parameter `RandomBound` applies when K is chosen to be $\mathbb{Q}$. Note that, although $\mathbb{Q}$ 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: Random Curves (ex-4c97b4)

```
> 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
```

124.3.6 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): CrvPln -> BoolElt, RngIntElt, RngMPol

```
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): CrvPln -> BoolElt, RngIntElt

Given a plane curve C defined over $\mathbb{Q}$ 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): Crv -> RngMPol`

Returns the (saturated) adjoint ideal of an ordinary plane curve C. If C is not ordinary then an error results.

`AdjointIdealForNodalCurve(C): Crv -> RngMPol`

`AdjointLinearSystemForNodalCurve(C, d): Crv, RngIntElt -> LinearSys`

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): RngMPol, RngIntElt -> LinearSys`

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): RngMPol, RngIntElt -> LinearSys`

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): Crv -> LinearSys`

`AdjointLinearSystem(C): Crv -> LinearSys`

`Adjoints(C, d): Crv, RngIntElt -> LinearSys`

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: Canonical Image of an Ordinary Plane Curve `(ex-d7505d)`

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
```

(continues on next page)

(continued from previous page)

```
> 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 +
```

(continues on next page)

(continued from previous page)

```
    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
```

124.4 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 *Rational Points and Point Sets* 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.

124.4.1 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 *Rational Points and Point Sets* of Chapter *Schemes* on schemes.

`C ! [a,...]: Crv, [RngElt] -> Pt`

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,...]: SetPt, [RngElt] -> Pt`

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): Pt -> Crv`

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): SetPt -> Crv`

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): Pt -> SeqEnum`

The sequence of ring elements corresponding to the coordinates of the point p.

`p[i]: Pt, RngIntElt -> RngElt`

`Coordinate``(p, i): Pt, RngIntElt -> SeqEnum`

The ith coordinate of the point p.

`p eq q: Pt, Pt -> BoolElt`

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): Pt -> Pt`

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.

124.4.2 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: Pt, Sch -> BoolElt`

`S in C: SeqEnum, Sch -> BoolElt`

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(p): Pt -> BoolElt`

`IsNonsingular(C, p): Sch, Pt -> BoolElt`

Returns `true` if and only if p is a nonsingular point of the curve C.

`IsSingular(p): Pt -> BoolElt`

`IsSingular(C, p): Sch, Pt -> BoolElt`

Returns `true` if and only if the point p is a singular point on the curve C.

`IsInflectionPoint(p): Pt -> BoolElt, RngIntElt`

`IsInflectionPoint(C, p): Sch, Pt -> BoolElt, RngIntElt`

`IsFlex(C, p): Sch, Pt -> BoolElt, RngIntElt`

`IsFlex(p): Pt -> BoolElt, RngIntElt`

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): Pt -> Crv`

`TangentLine(C, p): Crv, Pt -> Crv`

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(p): Pt -> Sch`

`TangentCone(C, p): Sch, Pt -> Sch`

The tangent cone to the curve C at the point p embedded in the same ambient space.

`IsTangent(C, D, p): Sch, Sch, Pt -> BoolElt`

Returns `true` if and only if the plane curves C and D are nonsingular and tangent at the point p.

124.4.3 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(p): Pt -> RngIntElt`

`Multiplicity(C, p): Sch, Pt -> RngIntElt`

The multiplicity of the curve C at the point p.

`IsDoublePoint(p): Pt -> BoolElt`

`IsDoublePoint(C, p): Crv, Pt -> BoolElt`

Returns `true` if and only if the point p is a double point of the curve C.

`IsOrdinarySingularity(p): Pt -> BoolElt`

`IsOrdinarySingularity(C, p): Sch, Pt -> BoolElt`

Returns `true` if and only if the point p is a singular point of the curve C with reduced tangent cone.

`IsNode(p): Pt -> BoolElt`

`IsNode(C, p): Sch, Pt -> BoolElt`

Returns `true` if and only if the point p is an ordinary double point of the curve C.

`IsCusp(p): Pt -> BoolElt`

`IsCusp(C, p): Crv, Pt -> BoolElt`

Returns `true` if and only if the point p is a nonordinary double point of the curve C.

`IsAnalyticallyIrreducible(p): Pt -> BoolElt`

`IsAnalyticallyIrreducible(C, p): CrvPln, Pt -> BoolElt`

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.

`DeltaAdjustment(C, p): Sch, Pt -> RngIntElt`

Argument 2, p, is a singular point in a pointset on argument 1, C, a 1-dimensional scheme. Returns the "delta adjustment" of p (or the set of conjugate singularities that p represents if it is defined over a finite extension of the base field of C), which is the amount that p contributes to the difference between the arithmetic genus of C and its geometric genus (i.e. the arithmetic genus of its normalisation). This adjustment is computed via local Hilbert series.

Example: Distinguishing Nodes and Cusps (ex-087199)

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
```

124.4.4 Resolution of Singularities

Again, all functions in this section only apply to plane curves.

`Blowup(C): CrvPln -> CrvPln, CrvPln`

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): CrvPln, Mtrx -> CrvPln, RngIntElt, RngIntElt`

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: Weighted Blowup (ex-bf0fac)

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 *Resolution Graphs and Splice Diagrams* 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 *Divisors*.

```
> Places(C ! Origin(A));
[
    Place at (0 : 0 : 1)
]
> Degree($1[1]);
1
```

124.4.5 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 $\mathbb{Q}$-Cartier divisor on V. Then the *log canonical threshold* (lct) of the log pair (V, D) at P is the number

$$\mathrm{lct}_P(V, D) = \sup\{\lambda \in \mathbb{Q} \mid (V, \lambda D) \text{ is log canonical at } P\} \in \mathbb{Q} \cup \{+\infty\}.$$

We can also consider the lct of D along the whole of V:

$$\begin{aligned}\mathrm{lct}(V, D) &= \inf\{\mathrm{lct}_P(V, D) \mid P \in V\} \\ &= \sup\{\lambda \in \mathbb{Q} \mid (V, \lambda D) \text{ is log canonical}\}.\end{aligned}$$

`LogCanonicalThreshold(C): Sch -> FldRatElt, BoolElt`

`LCT(C): Sch -> FldRatElt, BoolElt`

The log canonical threshold of the curve C computed at its singular k-points, where k is the base field of C.

`LogCanonicalThresholdAtOrigin(C): Sch -> FldRatElt`

The local log canonical threshold of the affine curve C computed at the origin.

`LogCanonicalThreshold(C, P): Sch, Pt -> FldRatElt`

`LCT(C, P): Sch, Pt -> FldRatElt`

The local log canonical threshold of the curve C computed at the point P.

`LogCanonicalThresholdOverExtension(C): Sch -> FldRatElt`

The log canonical threshold of the curve C computed at all singular points including those defined over some base field extension.

Example: Log Canonical Thresholds of Plane Cubics `(ex-abc557)`

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]) in PrimeComponents(D);
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: Log Canonical Threshold over an Extension (ex-bbea7a)

Here we exhibit a curve C over the rationals, $\mathbb{Q}$, that has singularities defined over a splitting field, k, where $\mathrm{lct}(C)$ (over k) $< \mathrm{lct}(C)$ (over $\mathbb{Q}$). 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 -
```

(continues on next page)

(continued from previous page)

```
>      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
```

124.4.6 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 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): Sch, Sch, Pt -> BoolElt

Returns `true` if and only if the point p lies on both curves C and D.

IsTransverse(C, D, p): Sch, Sch, Pt -> BoolElt

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): Sch, Sch, Pt -> RngIntElt

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): CrvPln, CrvPln -> List

IntersectionNumbers(F, G): RngMPolElt, RngMPolElt -> List

```
Global: BoolElt                    Default: false
```

These intrinsics 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 k: a finite field, an algebraic field, or 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 k, which represents a Galois conjugacy class of $([K : k])$ 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 $P = k[x, y, z]$. 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) $[*f(y), 1, 0*]$ with $f(y)$ an irreducible polynomial in $k[y]$. 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 $k[y]$ and $h(x, y)$ a polynomial in $k[x, y]$ whose image in $(k[y]/(g(y)))[x]$ 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 $h(x, \beta)$.

The parameter `Global` applies to the polynomial version. If it takes its default value `false`, both the two-variable and one-variable polynomial rings used in the type ii) and iii) representations will be non-global versions with the y and x, y labelling of variables as shown. This may have the disadvantage that elements returned by *different* calls to the intrinsic cannot be directly compared because they lie in different rings. If `Global` is `true`, the global one- and two-variable polynomial rings are used but in this case the variables are not labelled by the intrinsic.

Example: Local Intersection of Tangent Curves `(ex-d39f91)`

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
```

Example: Intersection Numbers from Hilmar and Smyth (ex-d12c4a)

This example is taken from the paper of Hilmar and Smyth. We use the polynomial version for which the output is more explicit.

```
> P<x,y,z> := PolynomialRing(Rationals(),3,"grevlex");
> A := (y-z)*x^5+(y^2-y*z)*x^4+(y^3-y^2*z)*x^3+(-y^2*z^2+y*z^3)*x^2+
>   (-y^3*z^2+y^2*z^3)*x-y^4*z^2+y^3*z^3;
> B := (y^2-2*z^2)*x^2+(y^3-2*y*z^2)*x+y^4-y^2*z^2-2*z^4;
> c := IntersectionNumbers(A,B);
> c;
[* <[* x + y, y^2 + 1, 1 *], 1>, <[* x - y^3, y^4 + 1, 1 *], 1>,
  <[* y^2 + y + 1, 1, 0 *], 2>, <[* x^2 + x + 2, y - 1, 1 *], 1>,
  <[* 1, 0, 0 *], 2>, <[* x^2 + x*y + 2, y^2 - 2, 1 *], 1>,
  <[* x^3 - y, y^2 - 2, 1 *], 1>  *]
```

124.5 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.

124.5.1 Genus and Singularities

`Genus(C): Crv -> RngIntElt`

`GeometricGenus(C): Crv -> RngIntElt`

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.

`GenusViaArithmeticGenus(C): Crv -> RngIntElt`

An alternative method for computation of the genus of C, which should be a projective curve. The method is to compute the (much easier) arithmetic genus of C and then to subtract off the "delta adjustments" over the set of singular points of C, which are computed with the `DeltaAdjustment` intrinsic. Requires that C is projective and lies in an ambient where its singular subscheme is computable.

`ArithmeticGenus(C): Crv -> RngIntElt`

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): CrvPln -> RngIntElt`

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): Sch -> SetIndx`

The singular points of the curve C which are defined over the base field of C.

`SingularPointsOverAlgebraicClosure(C): Sch -> SetIndx`

The singular points of the curve C over an algebraic closure of its base field. The singular locus of C must be zero dimensional. The base field must be the rational field, a finite field, or an algebraically closed field. `SingularPointsOverSplittingField` returns the same points over a number field when the base field is the rational field, and over an explicit finite extension when it is a finite field.

`SingularPointsOverSplittingField(C): Sch -> SetIndx`

```
Absolute: BoolElt                    Default: false
```

The singular points of the curve C over a splitting field, a finite extension of the base field, as computed by `PointsOverSplittingField`. The singular locus of C must be zero dimensional. Over the rational field the splitting field is a number field, or the rational field itself when no extension is needed. By default the number field is a tower. If the parameter `Absolute` is set to `true`, it is an absolute number field instead.

`HasSingularPointsOverExtension(C): Sch -> BoolElt`

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): Sch -> Sch`

`InflectionPoints(C): Sch -> Sch`

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: Sch, Sch -> BoolElt`

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): Sch, Sch -> BoolElt`

Returns `true` if and only if the curve C is contained (scheme-theoretically) in the curve D.

Example: Genus of a Cuspidal Cubic `(ex-74b2f1)`

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
```

124.5.2 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): Sch -> Sch`

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): Sch -> Sch`

The closure of the curve C in the projective closure of its ambient affine space.

Example: Uniqueness of Projective Closure (ex-d4f8cb)

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): Aff -> CrvPln`

The line which is the complement of the affine plane A embedded in the projective closure of A.

`PointsAtInfinity(C): Crv -> SetEnum`

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): Crv, RngIntElt -> SeqEnum`

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: Second Affine Patch (ex-caf4e0)

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
```

124.5.3 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): Crv -> BoolElt

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): Crv -> BoolElt

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): Crv -> CrvEll, MapSch

EllipticCurve(C, p): Crv, Pt -> CrvEll, MapSch

EllipticCurve(C, p): Crv, PlcCrvElt -> CrvEll, MapSch

See the description of EllipticCurve(C): Sch → CrvEll, MapSch in Chapter Elliptic Curves.

IsHyperelliptic(C): Crv -> BoolElt, CrvHyp, MapSch

Eqn: BoolElt	Default: true

IsGeometricallyHyperelliptic(C): Crv -> BoolElt, Crv, MapSch

Map : BoolElt	Default: true
IsHyp: Verbose	Default: 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: Testing for Hyperellipticity (ex-3920fc)

```
> 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
> ] );
```

(continues on next page)

(continued from previous page)

```
> 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
```

124.6 Maps and Curves

124.6.1 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 *Schemes* on schemes for more details about maps.

IdentityAutomorphism(A): Sch -> AutSch

Translation(A, p): Sch, Pt -> AutSch

FlipCoordinates(A): Sch -> AutSch

Automorphism(A, q): Sch, RngMPolElt -> AutSch

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): Crv, Pt -> Crv, AutSch

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: Translation to Infinity (ex-cb63a5)

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): MapSch, Pt -> Pt`

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: Evaluating a Map by Power Series (ex-60d1f5)

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
```

124.6.2 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` maps and to the covariant maps ϕ_*, corresponding to the Norm between the function fields, as `Pushforward` maps. Divisor groups and other function field related items are discussed in Section *Function Fields*.

`Degree(m): MapSch -> RngIntElt`

Returns the degree of a non-constant dominant map m between curves.

`RamificationDivisor(m): MapSch -> DivCrvElt`

Returns the ramification divisor of a non-constant dominant map m between irreducible curves.

`Pullback(phi, X): MapSch, FldFunFracSchElt -> FldFunFracSchElt`

`Pullback(phi, X): MapSch, DiffCrvElt -> DiffCrvElt`

`Pullback(phi, X): MapSch, DivCrvElt -> DivCrvElt`

`Pullback(phi, X): MapSch, PlcCrvElt -> DivCrvElt`

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): MapSch, FldFunFracSchElt -> FldFunFracSchElt`

`Pushforward(phi, X): MapSch, PlcCrvElt -> DivCrvElt`

`Pushforward(phi, X): MapSch, DivCrvElt -> DivCrvElt`

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: Pushforward and Pullback by a Map (ex-67ce2e)

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
```

124.7 Automorphism Groups of Curves

Magma now (from 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 General Automorphisms.

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).

124.7.1 Group Creation Functions

`AutomorphismGroup(C): Crv -> GrpAutCrv`

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): Crv, SeqEnum -> GrpAutCrv`

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): Crv -> SeqEnum`

```
Bound: RngIntElt                    Default: Infinity()
```

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): Crv, Crv -> BoolElt, MapSch`

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): Crv, Crv -> SeqEnum`

`Bound: RngIntElt`	`Default: Infinity()`

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`.

124.7.2 Automorphisms

`A . i: GrpAutCrv, RngIntElt -> GrpAutCrvElt`

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): GrpAutCrv -> GrpAutCrvElt`

`Id(A): GrpAutCrv -> GrpAutCrvElt`

`A ! 1: GrpAutCrv, RngIntElt -> GrpAutCrvElt`

The identity element of the automorphism group A.

`A ! f: GrpAutCrv, MapSch -> GrpAutCrvElt`

`A ! f: GrpAutCrv, GrpAutCrvElt -> GrpAutCrvElt`

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): GrpAutCrvElt -> RngIntElt`

The order of the curve automorphism f.

`Inverse(f): GrpAutCrvElt -> GrpAutCrvElt`

The inverse of the curve automorphism f.

`f * g: GrpAutCrvElt, GrpAutCrvElt -> GrpAutCrvElt`

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: GrpAutCrvElt, RngIntElt -> GrpAutCrvElt`

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: GrpAutoElt, GrpAutoElt -> BoolElt`

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: GrpAutoElt, GrpAutoElt -> BoolElt`

The logical negation of the preceding function.

`SchemeMap(f): GrpAutCrvElt -> MapAutSch`

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`.

124.7.3 Automorphism Group Operations

`Curve(A): GrpAutCrv -> Crv`

The curve of which A is a group of automorphisms.

`Order(A): GrpAutCrv -> RngIntElt`

The order of A.

`FactoredOrder(A): GrpAutCrv -> [ <RngIntElt, RngIntElt> ]`

The factored order of A.

`NumberOfGenerators(A): GrpAutCrv -> RngIntElt`

`Ngens(A): GrpAutCrv -> RngIntElt`

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): GrpAutCrv -> SeqEnum`

The generators of A - a small set of `GrpAutCrvElt` elements of A, which generate it as a group - returned in a sequence.

`PermutationGroup(A): GrpAutCrv -> GrpPerm`

The permutation group of A which gives the abstract group representation of the automorphism group.

`PermutationRepresentation(A): GrpAutCrv -> GrpPerm, Map`

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): GrpAutCrv -> Grpmat, Map, SeqEnum`

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: GrpAutCrvElt, GrpAutCrv -> BoolElt`

`phi in A: MapSch, GrpAutCrv -> BoolElt`

Returns whether a curve automorphism, given as a `GrpAutCrvElt` or `MapSch`, is equal to an automorphism lying in A.

`A subset B: GrpAutCrv, GrpAutCrv -> BoolElt`

Returns whether A is actually a subgroup of B as automorphism groups of the same curve.

124.7.4 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 *Maps Induced by Morphisms*.

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): GrpAutCrvElt, Pt -> Pt`

`f(X): GrpAutCrvElt, FldFunFracSchElt -> FldFunFracSchElt`

`f(X): GrpAutCrvElt, PlcCrvElt -> PlcCrvElt`

`f(X): GrpAutCrvElt, DivCrvElt -> DivCrvElt`

`f(X): GrpAutCrvElt, DiffCrvElt -> DiffCrvElt`

`X @ f: Pt, GrpAutCrvElt -> Pt`

`X @ f: FldFunFracSchElt, GrpAutCrvElt -> FldFunFracSchElt`

`X @ f: PlcCrvElt, GrpAutCrvElt -> PlcCrvElt`

`X @ f: DivCrvElt, GrpAutCrvElt -> DivCrvElt`

`X @ f: DiffCrvElt, GrpAutCrvElt -> DiffCrvElt`

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: FldFunFracSchElt, GrpAutCrvElt -> FldFunFracSchElt`

`X @@ f: PlcCrvElt, GrpAutCrvElt -> PlcCrvElt`

`X @@ f: DivCrvElt, GrpAutCrvElt -> DivCrvElt`

`X @@ f: DiffCrvElt, GrpAutCrvElt -> DiffCrvElt`

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: Automorphisms of the Fermat Quartic (ex-88cd9b)

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: Automorphisms of a Superelliptic Curve (ex-8b9b3c)

In the next example we look at a superelliptic curve over $GF(7^2)$ with isomorphism group a central extension of $PGL_2(\mathbb{F}_7)$ 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: Isomorphic Models of the Curve X(7) (ex-a782ea)

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;
```

(continues on next page)

(continued from previous page)

```
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 $PSL_2(\mathbb{F}_7)$) 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
```

124.7.5 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): GrpAutCrv -> Crv, MapSch`

`CurveQuotient(L): SeqEnum[MapAutSch] -> Crv, MapSch`

Example: Atkin-Lehner Quotients of X0(54) (ex-3ef244)

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 +
```

(continues on next page)

(continued from previous page)

```
    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
```

The argument 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. A subgroup of G can be specified by a list of generators.

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:

$C \to C/G$ 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: Quotient of the Klein Quartic `(ex-e1fc9c)`

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 a 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
```

(continues on next page)

(continued from previous page)

```
> 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.

124.8 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 *Function Fields and their Elements*.

124.8.1 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): Crv -> FldFunFracSch`

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): Crv -> BoolElt

Curve(F): FldFunFracSch -> Crv

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: FldFunFracSch, RngElt -> FldFunFracSchElt

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): FldFunFracSchElt -> RngFunFracElt

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: Creating a Function Field (ex-0441b0)

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: Pt, FldFunFracSchElt -> RngElt

f(p): Pt, FldFunFracSchElt -> RngElt

Evaluate(f, p): RngElt, Pt -> RngElt

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): FldFunFracSchElt[Crv], PlcCrvElt -> RngSerElt, FldFunFracSchElt

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): FldFunFracSch[Crv], PlcCrvElt -> RngSer, Map

```
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): FldFunFracSchElt[Crv] -> RngIntElt

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): RngElt, Pt -> RngIntElt

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): Pt -> Map

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): Pt -> FldFunFracSchElt

A rational function on the curve of the nonsingular point p which having valuation 1 at p.

Module(S): [FldFunFracSchElt[Crv]] -> Mod, Map, [ModElt]

```
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): [FldFunFracSchElt[Crv]] -> ModTupRng

Relations(S, m): [FldFunFracSchElt[Crv]], RngIntElt -> ModTupRng

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): Crv -> RngIntElt

The genus of the curve C.

FieldOfGeometricIrreducibility(C): Crv -> Rng, Map

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): Crv -> BoolElt

Returns `true` if the field of geometric irreducibility of the curve C is the base ring of C.

DimensionOfFieldOfGeometricIrreducibility(C): Crv -> RngIntElt

The dimension of the field of geometric irreducibility of the curve C over the base ring of C.

Example: Elements of a Function Field (ex-824936)

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): Crv -> [RngIntElt]

The gap numbers of the curve C.

WronskianOrders(C): Crv -> [RngIntElt]

The Wronskian orders of the curve C.

NumberOfPlacesOfDegreeOverExactConstantField(C, m): Crv[FldFin], RngIntElt -> RngIntElt

NumberOfPlacesDegECF(C, m): Crv[FldFin], RngIntElt -> RngIntElt

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): Crv[FldFin] -> RngIntElt

NumberOfPlacesOfDegreeOneECF(C): Crv[FldFin] -> RngIntElt

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): Crv[FldFin], RngIntElt -> RngIntElt

NumberOfPlacesOfDegreeOneECF(C, m): Crv[FldFin], RngIntElt -> RngIntElt

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): Crv -> RngIntElt

NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(C): Crv[FldFin] -> RngIntElt

NumberOfPlacesOfDegreeOneECFBound(C, m): Crv[FldFin], RngIntElt -> RngIntElt

NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(C, m): Crv[FldFin], RngIntElt -> RngIntElt

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): Crv[FldFin] -> DivCrvElt

Return a divisor of the curve C of degree 1 over its field of geometric irreducibility.

SerreBound(C): Crv[FldFin] -> RngIntElt

SerreBound(C, m): Crv[FldFin], RngIntElt -> RngIntElt

IharaBound(C): Crv[FldFin] -> RngIntElt

IharaBound(C, m): Crv[FldFin], RngIntElt -> RngIntElt

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.

124.8.2 Zeta Functions of Curves

The zeta function of a curve C over a finite field k is a generating function encoding $\#C(l)$ for all finite field extensions l/k. Determining the zeta function is equivalent to determining $\#C(k_i)$ for the field extensions k_i/k of degree $1 \leq i \leq g$, where g is the genus of C.

The algorithms available in Magma fall into three categories:

i) naive enumeration methods

ii) methods implemented specifically for genus 1 – see Chapter Elliptic Curves

iii) p-adic cohomology methods, which generalize Kedlaya's algorithm.

iv) variants of the algorithm in [Kyn22] that are based on Harvey's elementary p-adic trace formula for hypersurfaces [Har15].

The third category includes the following:

i) Harrison's extension of Kedlaya's algorithm for all hyperelliptic curves ([Har12]), and Vercauteren's implementation for characteristic 2

ii) Mestre's canonical lift method as adapted by Lercier and Lubicz, for characteristic 2

iii) (since V2.23) Minzlaff's implementation for superelliptic curves ([Min13]) which implements Harvey's $O(\sqrt{p})$ complexity improvement

iv) (since V2.23) Tuitman's algorithm for general curves ([Tui16] and [Tui17]).

While Tuitman's algorithm is the most general in this category, it requires as input a "good" lift to characteristic zero of the curve. For genus less than or equal to 5, such a model is given by `GonalityPreservingLift`, and this is done automatically by `ZetaFunction(C)`. For higher genus, there is no general algorithm for finding a good lift for Tuitman's method; however, for many curves it is trivial to find a good lift in practice. (For nondegenerate curves, an arbitrary lift to characteristic 0 will usually work. Any plane model of a curve over the rationals is a good lift for all but a finite number of primes.) In this case, Tuitman's algorithm may be invoked directly with `ZetaFunction(f, p)` where f defines the good lift – details below.

The algorithm outlined in [Kyn22] can be applied to any geometrically irreducible plane curve for which a plane model is provided, and thus can handle inputs beyond the current range of applicability of Tuitman's algorithm. (For more information on what inputs cannot be feasibly handled by Tuitman's algorithm or other Kedlaya-style algorithms, see Chapters 2 and 8 of [Kyn25].) Two variants of this algorithm have been implemented, one of which has time complexity that is quasilinear in the characteristic p and the other having time complexity that is quasiquadratic in p and which performs better in small characteristic.

```
LPolynomial(C): Crv[FldFin] -> RngUPolElt
LPolynomial(C, m): Crv[FldFin], RngIntElt -> RngUPolElt
ZetaFunction(C): Crv[FldFin] -> FldFunRatUElt
ZetaFunction(C, m): Crv[FldFin], RngIntElt -> FldFunRatUElt
```

```
Al           : MonStgElt                    Default: "Default"
ZetaFunction: Verbose                       Default:
```

The L-polynomial or ζ function of the projective normalisation of the curve C over the extension of degree m of the base ring of C, which must be a finite field.

By default, the algorithm is chosen automatically from the methods described above. The user may specify the algorithm by setting the optional argument `Al` to `"Naive"`, `"Harrison"`, `"Minzlaff"`, `"Tuitman"` or `"Kyng"`. (If the specified algorithm is not applicable, then the default algorithm may be used instead, or a runtime error may be thrown.)

Tuitman's Algorithm

The input to Tuitman's algorithm is a "good" lift of the desired curve to characteristic zero. The precise condition required is *Assumption 1* in [Tui17]. Basically, a good lift is a plane model whose reduction is birational to the given curve, and for which the 'x' coordinate map has branch and ramification locus which are smooth divisors over the ring of p-adic integers.

`GonalityPreservingLift(C): Crv[FldFun] -> RngUPolElt, AlgMatElt, AlgMatElt`

For a non-hyperelliptic curve of genus 3,4 or 5 over a finite field, this returns a good lift in the sense above. In particular, the lift has the same gonality as the given curve. The function returns a polynomial f defining the lift, over the rationals or a number field. It also returns matrices `W0, Winf` that can be passed as optional arguments to `ZetaFunction(f, p)` below, in order to save recomputing them.

The algorithms are described in [CT17]. The method involves randomized search. If it fails to find a lift (after a large number of trials) an error results. In such cases, it is highly probable that no lift exists, which happens for some curves over small fields.

`ZetaFunction(f, p): RngUPolElt, RngIntElt -> FldFunRatUElt`

```
W0        : AlgMatElt                    Default:
Winf      : AlgMatElt                    Default:
exactcoho: BoolElt                       Default: false
N         : RngIntElt                    Default:
Tuitman   : Verbose                      Default:
```

This directly invokes Tuitman's algorithm. The polynomial f, together with optional arguments `W0, Winf`, specify a curve over a field K which is either $\mathbb{Q}$ or a number field. The prime integer p must be inert in K, and the function returns the zeta function of the curve over the residue field Z_K/pZ_K. Note f is given as a (*univariate*) polynomial in $K[x][y]$ that is monic in y.

The optional arguments `W0, Winf` are matrices over $K(x)$ that define integral bases for the function field of the curve defined by f over $K[x]$ and $K[1/x]$, respectively. The other two options give some control over the internals. If a positive value is given for `N`, then calculations will be done with p-adic precision at most N – this may get the correct answer faster. If `exactcoho` is set to `true`, then cohomology is computed using exact linear algebra over K instead of p-adic linear algebra – this may be used as a check.

Zeta Function of a Singular Curve

`ZetaFunctionOfCurveModel(C): Crv[FldFin] -> FldFunRatUElt`

```
Zfn: FldFunRatUElt                       Default:
```

Returns the ζ function of the actual curve model C rather than that of its projective normalisation $\tilde{C}$. C must still be a reduced and irreducible curve and defined over a finite field $\mathbf{F}_q$ but it may be affine or singular.

The ζ function of C is defined by the usual formula in terms of the numbers $\#C(\mathbf{F}_{q^n})$ (see Appendix C of [Har77]) and is still a rational function, even if C is affine and/or singular. From the definition, it is easy to see that the ζ function of C differs from that of $\tilde{C}$ as follows. We multiply by $1-t^d$ to remove the contribution of a degree d place at infinity or lying over singular points of C and divide by $1-t^d$ to put back in the contribution of a Galois-conjugate set of singular points defined over $\mathbf{F}_{q^d}$.

The computation is performed in the obvious way, computing the places at infinity (in the affine case) and those over the singular locus using the standard function field machinery. The major work is in the initial computation of the ζ function of $\tilde{C}$, where `ZetaFunction` is called. If this ζ function is already known, it can be input via the parameter `Zfn`.

Example: Zeta Function of an Affine Weierstrass Model (ex-623718)

A simple example is where C is the usual affine Weierstrass model of an elliptic curve E. The curve C is non-singular and only differs from E by the removal of the single k-rational point O at infinity. The ζ function of C is therefore the usual ζ function of E multiplied by $1 - t$.

```
> k := GF(7);
> E := EllipticCurve([k|0,5]);
> ze1 := ZetaFunction(E);
> ze1;
(7*$.1^2 - $.1 + 1)/(7*$.1^2 - 8*$.1 + 1)
> A<x,y> := AffineSpace(k,2);
> C := Curve(A,y^2-x^3-5);
> ze2 := ZetaFunctionOfCurveModel(C);
> ze2;
(-7*$.1^2 + $.1 - 1)/(7*$.1 - 1)
> ze2/ze1;
-$.1 + 1
```

Point Counts, Traces, and the L-Polynomial

The L-polynomial of a genus g curve over a finite field $\mathbf{F}_q$ is determined by the Frobenius traces t_i, equivalently the point counts $\#C(\mathbf{F}_{q^i}) = q^i + 1 - t_i$, over the first g extensions $\mathbf{F}_q, \mathbf{F}_{q^2}, \ldots, \mathbf{F}_{q^g}$. The intrinsics below convert between these data and the L-polynomial in both directions, by Newton's identities together with the functional equation of the L-polynomial.

They do not depend on any zeta function algorithm, so they provide a cheap independent check on a computed `LPolynomial` or `ZetaFunction`: enumerate a few point counts directly and compare with the prediction.

`LPolynomialFromTracesOfFrobenius(t, q): SeqEnum[RngIntElt], RngIntElt -> RngUPolElt`

`LPolynomialFromNumbersOfPoints(n, q): SeqEnum[RngIntElt], RngIntElt -> RngUPolElt`

Given the Frobenius traces `t`, respectively the point counts `n`, of a genus g curve over $\mathbf{F}_q, \mathbf{F}_{q^2}, \ldots, \mathbf{F}_{q^g}$, return the corresponding L-polynomial, an integer polynomial of degree $2g$. The prime power `q` is the size of the base field, the i-th entry of the sequence refers to the extension $\mathbf{F}_{q^i}$, and the two forms of input are related by $\#C(\mathbf{F}_{q^i}) = q^i + 1 - t_i$. An empty sequence yields the genus 0 L-polynomial 1.

`TracesOfFrobeniusFromLPolynomial(L): RngUPolElt -> SeqEnum[RngIntElt], RngIntElt`

`NumbersOfPointsFromLPolynomial(L): RngUPolElt -> SeqEnum[RngIntElt], RngIntElt`

```
d: RngIntElt                    Default: 0
```

Given the L-polynomial `L` of a genus g curve, an integer polynomial of even degree $2g$ whose leading coefficient is q^g for a prime power q, return the sequence of Frobenius traces, respectively point counts, over $\mathbf{F}_q, \mathbf{F}_{q^2}, \ldots, \mathbf{F}_{q^g}$, together with q. The degree $2g$ must be positive: the genus 0 case, where q cannot be recovered from `L`, is not accepted. If the optional argument `d` is set to a positive integer then the first `d` terms are returned; `d` may exceed g because the full L-polynomial determines the traces over every finite extension.

Example: L-Polynomial from Point Counts (ex-40a015)

We recover the L-polynomial of a genus 2 curve over $\mathbf{F}_5$ from its point counts, check it against `LPolynomial`, and convert back.

```
> P<x> := PolynomialRing(GF(5));
> C := HyperellipticCurve(x^5 + x + 1);
> Genus(C);
2
> counts := [#Points(BaseChange(C, GF(5^i))) : i in [1..2]];
```

(continues on next page)

(continued from previous page)

```
> counts;
[ 6, 46 ]
> L := LPolynomialFromNumbersOfPoints(counts, 5);
> L;
25*T^4 + 10*T^2 + 1
> L eq LPolynomial(C);
true
> n, q := NumbersOfPointsFromLPolynomial(L);
> n;
[ 6, 46 ]
> q;
5
> tr := TracesOfFrobeniusFromLPolynomial(L);
> tr;
[ 0, -20 ]
> m := NumbersOfPointsFromLPolynomial(L : d := 1);
> m;
[ 6 ]
```

124.8.3 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 Algebraic Function Fields. 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): FldFunFracSch -> FldFun, Map

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): PlcCrvElt -> PlcFunElt

CurvePlace(C, p): Crv, PlcFunElt -> PlcCrvElt

FunctionFieldDivisor(d): DivCrvElt -> DivFunElt

CurveDivisor(C, d): Crv, DivFunElt -> DivCrvElt

FunctionFieldDifferential(d): DiffCrvElt -> DiffFunElt

CurveDifferential(C, d): Crv, DiffFunElt -> DiffCrvElt

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.

124.8.4 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.

Creation of Differentials

`DifferentialSpace(C): Crv -> DiffCrv`

The space of differentials of the curve C.

`SpaceOfDifferentialsFirstKind(C): Crv -> ModFld, Map`

`SpaceOfHolomorphicDifferentials(C): Crv -> ModFld, Map`

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): Crv -> [DiffCrvElt]`

`BasisOfHolomorphicDifferentials(C): Crv -> [DiffCrvElt]`

Given a curve C, this function returns a basis for the space of holomorphic differentials of C.

`DifferentialSpace(D): DivCrvElt -> ModFld, Map`

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): DivCrvElt -> [DiffCrvElt]`

Given a divisor D on a curve, this function returns the basis of the differential space of D.

`Differential(a): FldFunFracSchElt -> DiffCrvElt`

The exact differential $d(a)$ of the function field element a.

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): DiffCrv -> DiffCrvElt`

The identity differential of the differential space S of a curve.

`Curve(S): DiffCrv -> Crv`

The curve for which S is the space of differentials.

`Curve(a): DiffCrvElt -> Crv`

The curve to which the differential a belongs.

`f * x: RngElt, DiffCrvElt -> DiffCrvElt`

`x * f: DiffCrvElt, RngElt -> DiffCrvElt`

`x + y: DiffCrvElt, DiffCrvElt -> DiffCrvElt`

`- x: DiffCrvElt -> DiffCrvElt`

`x - y: DiffCrvElt, DiffCrvElt -> DiffCrvElt`

`x / r: DiffCrvElt, RngElt -> DiffCrvElt`

`x / y: DiffCrvElt, DiffCrvElt -> FldFunFracSchElt`

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: DiffCrv, DiffCrv -> BoolElt`

Returns `true` if and only if the two spaces of differentials S and T are the same.

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

Returns `true` if and only if the differentials a and b are equal.

`a in S: Any, DiffCrv -> BoolElt`

Returns `true` if and only if a is an element of the differential space S of a curve.

`IsExact(a): DiffCrvElt -> BoolElt`

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): DiffCrvElt -> BoolElt`

Returns `true` if and only if the differential a is the zero differential.

`Valuation(d, P): DiffCrvElt, PlcCrvElt -> RngIntElt`

The valuation of the differential d of a curve at the place P of the same curve.

`Residue(d, P): DiffCrvElt, PlcCrvElt -> RngElt`

The residue of the differential d of a curve at the place P of the same curve.

`Divisor(d): DiffCrvElt -> DivCrvElt`

The divisor $(f) + (dx)$ of the differential $d = f\,dx$ of a curve.

`Module(L): [DiffCrvElt] -> Mod, Map, [ ModElt ]`

```
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): [DiffCrvElt] -> ModTupFld`

`Relations(L, m): [DiffCrvElt], RngIntElt -> ModTupFld`

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): DiffCrvElt -> DiffCrvElt`

`Cartier(a, r): DiffCrvElt, RngIntElt -> DiffCrvElt`

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): Crv -> AlgMatElt, SeqEnum[DiffCrvElt]`

`CartierRepresentation(C, r): Crv, RngIntElt -> AlgMatElt, SeqEnum[DiffCrvElt]`

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 $r \in \mathbb{Z}^{\geq 1}$. 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: Holomorphic Differentials of a Plane Quartic `(ex-bb3bd8)`

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
```

124.9 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 *Linear Equivalence of Divisors*, 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 Algebraic Function Fields. A translation is made in the background for all functions described here. For information about function fields, their representations, and translations see Section *Function Fields*. The translations should not be explicitly needed in the ordinary course of working with curves and their divisors.

Note, however, that divisors of function fields have 4 possible representations, each having their own advantages and disadvantages. These representations of the *FunctionFieldDivisor* of a curve divisor are stated in Section 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 *Linear Equivalence of Divisors* 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.

124.9.1 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.

Sets of Places

`Places(C): Crv -> PlcCrv`

The set of places of the curve C.

`Curve(P): PlcCrv -> Crv`

The curve on which the places in the set of places P lie. This will be a projective curve.

`P eq Q: PlcCrv, PlcCrv -> BoolElt`

`P ne Q: PlcCrv, PlcCrv -> BoolElt`

Returns `true` if and only if the sets of places P and Q are (not) equal.

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): Crv[FldFin], RngIntElt -> SeqEnum`

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): Crv[FldFin], RngIntElt -> BoolElt, PlcCrvElt`

`RandomPlace(C, m): Crv[FldFin], RngIntElt -> BoolElt, PlcCrvElt`

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): Pt -> PlcCrvElt`

The place corresponding to the nonsingular point p of some curve.

`Places(p): Pt -> SeqEnum`

A sequence containing the places corresponding to the point p of some curve.

`Place(C, I): Crv, RngMPol -> PlcCrvElt`

Return the place of the curve C defined by the ideal I of the coordinate ring of C.

`WeierstrassPlaces(C): Crv -> [PlcCrvElt]`

The Weierstrass places of the curve C, which are the Weierstrass places of the zero divisor of C.

`WeierstrassPoints(C): Crv -> Sch`

The Weierstrass points of the curve C as a cluster.

`Place(Q): [FldFunFracSchElt] -> PlcCrvElt`

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): PlcCrvElt -> RngMPol`

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): PlcCrvElt -> FldFunFracSchElt, FldFunFracSchElt`

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: Places from Rational Functions (ex-52e224)

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): FldFunFracSchElt[Crv] -> SeqEnum[PlcCrvElt]`

`Poles(f): FldFunFracSchElt[Crv] -> SeqEnum[PlcCrvElt]`

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): Crv, RngElt -> [PlcCrvElt]`

`Poles(C, f): Crv, RngElt -> [PlcCrvElt]`

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): [FldFunFracSchElt[Crv]] -> [PlcCrvElt]`

`CommonZeros(C, L): Crv, [FldFunFracSchElt] -> [PlcCrvElt]`

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: Zeros and Poles (ex-8f7c99)

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> := FieldOfFractions(CoordinateRing(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: PlcCrvElt, PlcCrvElt -> DivCrvElt`

`- p1: PlcCrvElt -> DivCrvElt`

`p1 - p2: PlcCrvElt, PlcCrvElt -> DivCrvElt`

`k * p: RngIntElt, PlcCrvElt -> DivCrvElt`

`p div k: PlcCrvElt, RngIntElt -> DivCrvElt`

`p mod k: PlcCrvElt, RngIntElt -> DivCrvElt`

`Quotrem(p1, k): PlcCrvElt, RngIntElt -> DivCrvElt, DivCrvElt`

`Curve(P): PlcCrvElt -> Crv`

The projective curve on which the place P lies.

`RepresentativePoint(P): PlcCrv -> Pt`

A representative point of the projective model of the curve underlying the place.

`P eq Q: PlcCrvElt, PlcCrvElt -> BoolElt`

`P ne Q: PlcCrvElt, PlcCrvElt -> BoolElt`

Returns `true` if and only if the two places are (not) the same.

`P in S: Any, PlcCrv -> BoolElt`

`P notin S: Any, PlcCrv -> BoolElt`

`Valuation(f, P): RngElt, PlcCrvElt -> RngIntElt`

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): PlcCrvElt -> Map`

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): DiffCrvElt, PlcCrvElt -> RngIntElt`

The valuation of the differential a at the place P.

`Residue(a, P): DiffCrvElt, PlcCrvElt -> RngElt`

The residue of the differential a at the degree 1 place P.

`UniformizingParameter(P): PlcCrvElt -> FldFunFracSchElt`

A function on the curve of the place P having valuation 1 at P.

`IsWeierstrassPlace(P): PlcCrvElt -> BoolElt`

`IsWeierstrassPlace(D, P): DivCrvElt, PlcCrvElt -> BoolElt`

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): PlcCrvElt -> Rng`

The residue class field of the place P.

`Evaluate(a, P): FldFunFracSchElt, PlcCrvElt -> RngElt`

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): RngElt, PlcCrvElt -> FldFunFracSchElt`

`Lift(i, P): Infty, PlcCrvElt -> FldFunFracSchElt`

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): PlcCrvElt -> RngIntElt`

The degree of the place P of a curve over the base ring of the curve of P.

`GapNumbers(C, P): Crv, PlcCrvElt -> [RngIntElt]`

`GapNumbers(P): PlcCrvElt -> [RngIntElt]`

The gap numbers of the curve C at the degree 1 place P.

`Parametrization(C, p): Crv, Pt -> MapSch`

`Parametrization(C, p): Crv, PlcCrvElt -> MapSch`

`Parametrization(C, p, P): Crv, PlcCrvElt, Crv -> MapSch`

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.

124.9.2 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): Crv -> DivCrv

The group of divisors of the curve C. This curve may be either an affine or projective curve.

Curve(Div): DivCrv -> Crv

The curve that was used to create the divisor group Div, or its projective model.

Div1 **eq** Div2: DivCrv, DivCrv -> BoolElt

Div1 **ne** Div2: DivCrv, DivCrv -> BoolElt

Returns **true** if and only the divisor groups $Div1$ and $Div2$ are (not) equal.

124.9.3 Creation of Divisors

DivisorGroup(D): DivCrvElt -> DivCrv

The divisor group in which the divisor D lies.

Curve(D): DivCrvElt -> Crv

The (projective) curve on which the divisor D lies.

Identity(D): DivCrv -> DivCrvElt

Id(D): DivCrv -> DivCrvElt

D ! 0: DivCrv, RngIntElt -> DivCrvElt

The zero divisor of the divisor group D of a curve.

Div ! p: DivCrv, PlcCrvElt -> DivCrvElt

Div ! p: DivCrv, Pt -> DivCrvElt

Divisor(p): PlcCrvElt -> DivCrvElt

Divisor(p): Pt -> DivCrvElt

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): DivCrv, SeqEnum -> DivCrvElt

Divisor(C, S): Crv, SeqEnum -> DivCrvElt

Divisor(S): [<PlcCrvElt, RngIntElt>] -> DivCrvElt

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: Reconstructing a Divisor from Its Data (ex-5ce339)

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);
> DivisorGroup(C);
Group of divisors of Curve over GF(17) defined by
x^5 + x^2*y^3 + 16*z^5
> 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>
```

(continues on next page)

(continued from previous page)

```
]
> 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): Crv, RngElt -> DivCrvElt

PrincipalDivisor(D, f): DivCrv, RngElt -> DivCrvElt

PrincipalDivisor(f): FldFunFracSchElt[Crv] -> DivCrvElt

Divisor(C, f): Crv, RngElt -> DivCrvElt

Divisor(D, f): DivCrv, RngElt -> DivCrvElt

Divisor(f): FldFunFracSchElt[Crv] -> DivCrvElt

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): DiffCrvElt -> DivCrvElt

The divisor of the curve C corresponding to the differential a of C.

Divisor(C, X): Crv, Sch -> DivCrvElt

Divisor(D, X): DivCrv, Sch -> DivCrvElt

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): Crv, Pt, Pt -> DivCrvElt

Divisor(D, p, q): DivCrv, Pt, Pt -> DivCrvElt

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): Crv, RngMPol -> DivCrvElt

Divisor(D, I): DivCrv, RngMPol -> DivCrvElt

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): DivCrvElt -> SeqEnum

The decomposition sequence of D as a sequence of tuples of the form **<place, multiplicity>** characterizing the divisor D.

Support(D): DivCrvElt -> SeqEnum, SeqEnum

The sequence of places in the support of D, followed by their sequence of multiplicities in D.

Example: Creating Divisors on a Curve (ex-68cd9e)

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);
> F := FunctionField(C);
> D := Divisor(C,F!(x/z));
> D;
Divisor of Curve over GF(7) defined by
```

(continues on next page)

(continued from previous page)

```
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): Crv -> DivCrvElt

A divisor in the canonical divisor class of the curve C.

RamificationDivisor(C): Crv -> DivCrvElt

The ramification divisor of the curve C.

124.9.4 Arithmetic of Divisors

D + E: DivCrvElt, DivCrvElt -> DivCrvElt

D + E: DivCrvElt, PlcCrvElt -> DivCrvElt

D + E: PlcCrvElt, DivCrvElt -> DivCrvElt

D + E: PlcCrvElt, PlcCrvElt -> DivCrvElt

- D: DivCrvElt -> DivCrvElt

- D: PlcCrvElt -> DivCrvElt

D - E: DivCrvElt, DivCrvElt -> DivCrvElt

D - E: DivCrvElt, PlcCrvElt -> DivCrvElt

D - E: PlcCrvElt, DivCrvElt -> DivCrvElt

D - E: PlcCrvElt, PlcCrvElt -> DivCrvElt

n * D: RngIntElt, DivCrvElt -> DivCrvElt

n * D: RngIntElt, PlcCrvElt -> DivCrvElt

D **div** n: DivCrvElt, RngIntElt -> DivCrvElt

D **div** n: PlcCrvElt, RngIntElt -> DivCrvElt

D **mod** n: DivCrvElt, RngIntElt -> DivCrvElt

D **mod** n: PlcCrvElt, RngIntElt -> DivCrvElt

Basic formal arithmetic of divisors; D and E are divisors (or places) and n is an integer.

Quotrem(D, n): DivCrvElt, RngIntElt -> DivCrvElt, DivCrvElt

The quotient and remainder on dividing the divisor D by the integer n.

Degree(D): DivCrvElt -> RngIntElt

The sum of coefficients of the divisor D multiplied by the degrees of the places of the corresponding components.

IsEffective(D): DivCrvElt -> BoolElt

IsPositive(D): DivCrvElt -> BoolElt

Returns **true** if and only if all coefficients of the divisor D are nonnegative.

Numerator(D): DivCrvElt -> DivCrvElt

Denominator(D): DivCrvElt -> DivCrvElt

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): DivCrvElt -> DivElt, DivElt

The minimal effective divisors A and B such that the equality of divisors $D = A - B$ holds.

Example: Decomposition of a Divisor (ex-768c79)

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);
> F := FunctionField(C);
> D := Divisor(C,F!(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: Any, DivCrv -> BoolElt

d **notin** D: Any, DivCrv -> BoolElt

D **eq** E: DivCrvElt, DivCrvElt -> BoolElt

D **ne** E: DivCrvElt, DivCrvElt -> BoolElt

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: DivCrvElt, DivCrvElt -> BoolElt

D **le** E: DivCrvElt, DivCrvElt -> BoolElt

D **gt** E: DivCrvElt, DivCrvElt -> BoolElt

D **ge** E: DivCrvElt, DivCrvElt -> BoolElt

AreLinearlyEquivalent(D, E): DivCrvElt, DivCrvElt -> BoolElt

Returns **true** if and only the divisors D and E are linearly equivalent.

`IsZero(D): DivCrvElt -> BoolElt`

Returns `true` if and only if all coefficients of the divisor D are zero.

`IsCanonical(D): DivCrvElt -> BoolElt, DiffCrvElt`

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): DivCrvElt, DivCrvElt -> DivCrvElt`

`Gcd(D1, D2): DivCrvElt, DivCrvElt -> DivCrvElt`

`GreatestCommonDivisor(D1, D2): DivCrvElt, DivCrvElt -> DivCrvElt`

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): DivCrvElt, DivCrvElt -> DivCrvElt`

`Lcm(D1, D2): DivCrvElt, DivCrvElt -> DivCrvElt`

`LeastCommonMultiple(D1, D2): DivCrvElt, DivCrvElt -> DivCrvElt`

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: Canonical Divisor (ex-235268)

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);
> F := FunctionField(C);
> D := Divisor(C,F!(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
```

124.9.5 Other Operations on Divisors

`Ideal(D): DivCrvElt -> RngMPol`

Given a divisor D of a curve C, returns the ideal of the coordinate ring of the ambient of C of coordinate functions which cuts out D. The ideal D must be effective.

`Cluster(D): DivCrvElt -> Clstr`

Given a divisor D of a curve C, returns the cluster (zero-dimensional scheme) which gives D as a subscheme of C. This is the scheme defined by `Ideal(D)`. The ideal D must be effective.

`Valuation(D, p): DivCrvElt, Pt -> DivCrvElt`

`Valuation(D, P): DivCrvElt, PlcCrvElt -> DivCrvElt`

The coefficient of the divisor summand of the divisor D corresponding to the point p or place P.

`ComplementaryDivisor(D, p): DivCrvElt, Pt -> DivCrvElt`

`ComplementaryDivisor(D, P): DivCrvElt, PlcCrvElt -> DivCrvElt`

The divisor after removing from the divisor D the component corresponding to the point p or place P.

124.10 Linear Equivalence of Divisors

124.10.1 Linear Equivalence and Class Group

`IsPrincipal(D): DivCrvElt -> BoolElt, FldFunFracSchElt`

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): DivCrvElt, DivCrvElt -> BoolElt`

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): DivCrvElt -> BoolElt, RngElt, RngIntElt`

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: Canonical Divisor as a Hyperplane Section (ex-819631)

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): Crv[FldFin] -> GrpAb, Map, Map`

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): Crv[FldFin] -> RngIntElt`

The order of the class group of the curve C.

`GlobalUnitGroup(C): Crv[FldFin] -> GrpAb, Map`

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: Divisor Class Group over a Finite Field (ex-f90495)

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;
```

(continues on next page)

(continued from previous page)

```
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): Crv[FldFin] -> [RngIntElt]

The abelian invariants of the curve C.

ClassGroupPRank(C): Crv[FldFin] -> RngIntElt

The p-rank of the class group of C.

HasseWittInvariant(C): Crv[FldFin] -> RngIntElt

The Hasse–Witt invariant of the curve C.

124.10.2 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): DivCrvElt -> DivCrvElt, RngIntElt, DivCrvElt, FldFunFracSchElt`

`Reduction(D, A): DivCrvElt, DivCrvElt -> DivCrvElt, RngIntElt, DivCrvElt, FldFunFracSchElt`

Let D be a divisor. Denote the result of both functions by $\tilde{D}$, r, A, and a (for the second function the input A always equals the output A). The divisor A has (must have) positive degree and the following holds:

(i) $D = \tilde{D} + rA - (a)$,

(ii) $\tilde{D} \geq 0$ and $\deg(\tilde{D}) < g + \deg(A)$ (over the field of geometric irreducibility),

(iii) $\tilde{D}$ has minimal degree among all such divisors satisfying (i), (ii).

`RiemannRochSpace(D): DivCrvElt -> ModFld, Map`

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): DivCrvElt -> SeqEnum`

A sequence containing a basis of the Riemann-Roch space $L(D)$ of the divisor D.

`ShortBasis(D): DivCrvElt -> SeqEnum`

A sequence containing a basis of the Riemann-Roch space $L(D)$ of the divisor D in short form.

`Dimension(D): DivCrvElt -> RngIntElt`

The dimension $\ell(d)$ of the Riemann-Roch space of the divisor D.

`DifferentialSpace(D): DivCrvElt -> ModFld, Map`

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): DivCrvElt -> SeqEnum`

A basis of the space of differentials of the divisor D.

`IndexOfSpeciality(D): DivCrvElt -> RngIntElt`

The index of speciality of the divisor D, that is the dimension $\ell(K_C - D)$ appearing in the Riemann–Roch formula.

`IsSpecial(D): DivCrvElt -> BoolElt`

Returns `true` if and only if the divisor D is special.

`GapNumbers(D): DivCrvElt -> SeqEnum`

`GapNumbers(D, p): DivCrvElt, PlcCrvElt -> SeqEnum`

The gap numbers of the divisor D, at the place p if included.

`GapNumbers(p): Pt -> SeqEnum`

The gap numbers of the nonsingular point p.

`WeierstrassPlaces(D): DivCrvElt -> SeqEnum`

`WeierstrassPoints(D): DivCrvElt -> SeqEnum`

A sequence containing the Weierstrass places (or underlying points) of the divisor D.

`WronskianOrders(D): DivCrvElt -> SeqEnum`

The Wronskian orders of the divisor D.

`RamificationDivisor(D): DivCrvElt -> DivCrvElt`

The ramification divisor of the divisor D.

`DivisorMap(D): DivCrvElt -> MapSch`

`DivisorMap(D, P): DivCrvElt, Prj -> MapSch`

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): Crv -> MapSch`

`CanonicalMap(C, P): Crv, Prj -> MapSch`

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): Crv, MapSch -> Crv, BoolElt`

`CanonicalImage(C, eqns): Crv, SeqEnum -> Crv, BoolElt`

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: Canonical Map (ex-6c8d01)

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
```

(continues on next page)

(continued from previous page)

```
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 *Trigonal Curves*.

124.10.3 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 [Die], 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 [Die] 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): DivCrvElt, DivCrvElt, DivCrvElt, RngIntElt -> RngIntElt

IndexCalculus(D1, D2, D0, np, n, rr): DivCrvElt, DivCrvElt, DivCrvElt, RngIntElt, RngIntElt, RngIntElt -> RngIntElt

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): DivCrvElt, DivCrvElt, DivCrvElt, RngIntElt, RngIntElt -> MtrxSprs, SeqEnum, SeqEnum, DivCrvElt, DivCrvElt, RngIntElt, RngIntElt

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): RngIntElt, DivCrvElt, DivCrvElt -> DivCrvElt`

Effective divisor E of minimal degree such that $E - \deg(E)D0$ is equivalent to $n(D - \deg(\mathrm{D})D0)$.

Example: Discrete Logarithm by Index Calculus (ex-f3220c)

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);
[ <175310448404461045718069548360655883 7, 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
```

(continues on next page)

(continued from previous page)

```
DLP mod 175310448404461045718069548360655837:
125220860410617991206452745026979 42
Time: 164.900
> MultiplyDivisor(D1,D1,D0) eq Reduction(D2,D0);
true
```

124.11 Advanced Examples

These examples are intended to demonstrate basic programming in Magma using the functions of this chapter together with a few from Chapter *Schemes*. 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.

124.11.1 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: Trigonal Curve (ex-3b68a5)

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
```

(continues on next page)

(continued from previous page)

```
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^1_3 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 |
>       [r^2*u, r*s*u, s^2*u, r^4*v, r^3*s*v, r^2*s^2*v, r*s^3*v, s^4*v] >;
> 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^1_3. 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^1_3. 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));
> P1 := Curve(Codomain(mpD));  // get P1 as a curve
> mpD := map<C->P1|[rat,1]>; mpD;
Mapping from: CrvPln: C to Crv: P1
with equations :
X
Z
```

124.11.2 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 Linear Codes over Finite Fields. 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: Code from the Klein Quartic (ex-d59801)

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.

```
> FF8 := FunctionField(C8);
> D := Divisor(C8, FF8!(a/c + b/c + 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 0 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 0 0 t8^2 t8^3 t8 t8^4]
```

124.12 Curves over Global Fields

124.12.1 Finding Rational Points

There are also various point search routines for more general schemes.

`PointsCubicModel(C, B : parameters): Crv, RngIntElt -> SeqEnum`

```
OnlyOne         : BoolElt                    Default: false
ExactBound      : BoolElt                    Default: false
CubicModelSearch: Verbose                    Default: 1
```

Given a plane cubic over the rationals, this function searches, by a reasonably efficient method due to Elkies [Elk], 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 $\mathbb{Z}$ will be found first. If `ExactBound` is set to `true`, then points that are found with height larger than B will be ignored.

Example: Rational Points on a Plane Cubic (ex-5a2291)

```
> 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
```

124.12.2 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.

Creation of Regular Models

`RegularModel(C, P): Crv, Any -> CrvRegModel`

```
RegularModel: Verbose                    Default: 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.

Using Regular Models

`IntersectionMatrix(M): CrvRegModel -> Mtrx`

Given a regular model, this returns a matrix whose entries are the intersection multiplicities of the (reduced, irreducible) components of the special fibre.

`Multiplicities(M): CrvRegModel -> SeqEnum`

Given a regular model, this returns a sequence giving the multiplicities of the components. The components of the model always come in the same order.

`ComponentGroup(M): CrvRegModel -> GrpAb`

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): CrvRegModel, Pt -> SeqEnum, SeqEnum, Tup`

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.

124.12.3 Minimization and Reduction for Plane Curves

Minimization and reduction is a search for a linear transformation, that leads to a nice equation. The first step is called minimization. It searches for linear changes of the coordinates that result in models with smaller invariants. This is done locally for each prime of sufficiently bad reduction. Then the reduction step searches for a unimodular transformation that results in smaller coefficients.

Here, we describe the routines for the minimization and reduction for plane curves of degree at least 3.

`MinRedTernaryForm(F): RngMPolElt -> RngMPolElt, AlgMatElt, FldRatElt`

`MinimizeReduce(C): CrvPln -> CrvPln, AlgMatElt`

`AdHocOnly: BoolElt`	`Default: false`

Given a plane curve C or equivalently a ternary form F, this function returns an isomorphic model with small coefficients and the transformation matrix used. The computation is done by calling the minimization and the reduction function described in this section.

Minimization for Plane Curves

`MinimizeTernaryFormAtp(F, p): RngMPolElt, RngIntElt -> RngMPolElt, AlgMatElt, RngIntElt`

`Minimisation: Verbose`	`Default: 3`

Given a plane curve represented by a ternary form, this function returns a form representing a p-minimal model of the curve and the transformation matrix used.

`MinimizeTernaryForm(F): RngMPolElt -> RngMPolElt, AlgMatElt, RngIntElt`

`AdHocRed: BoolElt`	`Default: false`

Given a plane curve represented by a ternary form, this function returns a form representing a model of the curve that is p-minimal for all primes p and the transformation matrix used. If `AdHocRed` is set, the function applies a reduction steps to intermediate results.

Reduction for Plane Curves

`ReducePlaneCurve(C): Crv -> Crv, Mtrx`

`ReducePlaneCurve(f): MPolElt -> RngMPolElt, Mtrx`

`ReduceTernaryForm(f): MPolElt -> RngMPolElt, Mtrx`

```
AdHocOnly: BoolElt                    Default: false
Reduction: Verbose                    Default: 3
```

Given a ternary form f or a plane curve C of degree > 2 with integer or rational coefficients. A new form or curve with smaller coefficients and the unimodular transformation matrix used are returend.

If `AdHocOnly` is set, only a quick and simple reduction is done. Otherwise, the routine computes a covariant cluster and applies cluster reduction to it.

Reduction of Point Clusters

The reduction of point clusters can be called directly as follows:

`ReduceCluster(X): SeqEnum -> SeqEnum, Mtrx, Mtrx`

`ReduceCluster(C): Clstr -> Clstr, Mtrx, Bool`

```
eps              : FldReElt                Default: 1e-6
c                : FldReElt                Default: 1
ClusterReduction: Verbose                  Default: 3
```

Here X is a sequence of n-dimensional vectors of complex numbers or C is a cluster of points in projective space. 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.

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. The algorithm is due to Stoll (see [Sto11]).

Example: Minimising and Reducing a Plane Quartic `(ex-77ecd8)`

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 + 6145946820119559*x^3*z -
11625449190228*x^2*y^2 - 4102113209795298*x^2*y*z + 16654003843623327 72*x^2*z^2
- 62468022936*x*y^3 - 417499281622764*x*y^2*z - 72808467360772908*x*y*z^2 +
2005588071197635933 2*x*z^3 - 16293798512*y^4 - 875035770696*y^3*z -
3749491014537304*y^2*z^2 - 430694749052979580*y*z^3 + 90567449117290511049*z^4
> MinRedTernaryForm(C2);
-3*x^4 + 10*x^3*y + 5*x^3*z + 6*x^2*y^2 - 7*x^2*y*z - 3*x^2*z^2 + 7*x*y^3 +
    9*x*y^2*z + 5*x*y*z^2 + 4*x*z^3 - 3*y^4 - 2*y^3*z - 9*y^2*z^2 - 6*y*z^3 +
    6*z^4
[-21564104   4611976     14878]
[  2804580    148866     -1935]
[  1197852   -256188      3330]
450309688534260450217470624641049 6
```

We do not get the initial curve, but we get a curve with coefficients of the same size.

124.13 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 [Har13]. 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.

124.13.1 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): Crv -> RngIntElt, BoolElt, MapSch`

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): Crv -> MapSch`

`CliffordIndexOne(C, X): Crv, Sch -> MapSch`

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: Trigonal Map of a Canonical Curve (ex-b24415)

```
> 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
```

124.13.2 Small Genus Functions

This section contains intrinsics to compute gonal maps for curves with genus greater than one and less than seven.

Genus2GonalMap(C): Crv -> MapSch

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): Crv -> RngIntElt, MapSch

```
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): Crv -> RngIntElt, MapSch`

```
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): Crv -> RngIntElt, MapSch, Crv, UserProgram`

```
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 extension 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): Crv -> RngIntElt, RngIntElt, MapSch, MapSch

```
DataOnly   : BoolElt                    Default: false
IsCanonical: BoolElt                    Default: false
```

For a genus 6 curve C, returns the gonality (2, 3, or 4), a second type identifier, 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: Gonal Maps of Genus 5 and 6 Curves (ex-431e07)

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 :
```

(continues on next page)

(continued from previous page)

```
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
```

124.13.3 Small Genus Plane Models

This section contains intrinsics to compute minimal degree birational plane models for curves of genus 5 or 6.

Genus6PlaneCurveModel(C): Crv -> BoolElt, MapSch

```
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 [Har13]). 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): Crv -> BoolElt, MapSch`

`Genus5PlaneCurveModel(C, P): Crv, Pt -> BoolElt, MapSch`

`Genus5PlaneCurveModel(C, Z): Crv, Sch -> BoolElt, MapSch`

`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: Plane Model of X0(58) (ex-df6704)

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],
```

(continues on next page)

(continued from previous page)

```
-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
```

(continues on next page)

(continued from previous page)

```
y
z + t
```

124.14 Bibliography

- [CT17] Wouter Castryck and Jan Tuitman. Point counting on curves using a gonality preserving lift. *The Quarterly Journal of Mathematics*, 2017.
- [Die] C. Diem. "An Index Calculus Algorithm for Plane Curves of Small Degree." 543–557.
- [Elk] N. Elkies. "Rational Points Near Curves and Small Nonzero $|x^3 - y^2|$ via Lattice Reduction." 33–63.
- [Ful69] William Fulton. *Algebraic Curves*. Volume of Mathematics Lecture Note Series. W. A. Benjamin, New York-Amsterdam, 1969.
- [Har13] M. C. Harrison. Explicit solution by radicals, gonal maps and plane models of algebraic curves of genus 5 or 6. *J. Symb. Comp.*, 51:3–21, 2013.
- [Har12] Michael C. Harrison. An extension of Kedlaya's algorithm for hyperelliptic curves. *J. Symbolic Comput.*, 47(1):89–101, 2012.
- [Har77] Robin Hartshorne. *Algebraic Geometry, GTM 52*. Springer, ASpringer, 1977.
- [Har15] David Harvey. Computing zeta functions of arithmetic schemes. *Proceedings of the London Mathematical Society*, 111(6):1379–1401, 2015. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/pdv056, arXiv:https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/pdv056, doi:https://doi.org/10.1112/plms/pdv056.
- [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.
- [Kyn22] Madeleine Kyng. Computing zeta functions of algebraic curves using Harvey's trace formula. *Res. Number Theory*, 8(4):Paper No. 100, 17, 2022. URL: https://doi.org/10.1007/s40993-022-00398-7, doi:10.1007/s40993-022-00398-7.
- [Kyn25] Madeleine Kyng. *Computing zeta functions of algebraic curves using Harvey's trace formula*. Dissertation, The University of New South Wales, May 2025. available at https://web.maths.unsw.edu.au/ davidharvey/research/kyng-thesis.pdf.
- [Min13] Moritz Minzlaff. *Frobenius-stable lattices in rigid cohomology of curves*. Dissertation, Technische Universität Berlin, 2013. URL: http://dx.doi.org/10.14279/depositonce-3566.
- [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 and others, editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of Springer Algorithms and Computation in Mathematics Series, 251–277. Springer-Verlag, 2002.
- [Sto11] "Reduction theory of point clusters in projective space."
- [Tui16] Jan Tuitman. Counting points on curves using a map to $\mathbf{P^1}$. *Math. Comp.*, 85(298):961–981, 2016.
- [Tui17] Jan Tuitman. Counting points on curves using a map to $\mathbf{P^1}$, II. *Finite Fields Appl.*, 45:301–322, 2017.
- [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.

Chapter 125

Resolution Graphs and Splice Diagrams

125.1 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.

125.2 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 $f\colon S \to P^2$, 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 of 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.

125.2.1 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 $1, \ldots, 6$ 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: GrphRes, GrphRes -> BoolElt`

Returns `true` if and only if the resolution graphs g and h are the same object in Magma.

`ResolutionGraphVertex(g, i): GrphRes, RngIntElt -> GrphResVert`

`g ! i: GrphRes, RngIntElt -> GrphResVert`

The vertex of the resolution graph g with index i.

`Vertex(v): GrphResVert -> GrphVert`

The underlying directed graph vertex of the resolution graph vertex v.

`ResolutionGraph(v): GrphResVert -> GrphRes`

The resolution graph of which v is a vertex.

`IsVertex(g, v): GrphRes, GrphResVert -> BoolElt`

Returns `true` if and only if v is a vertex of the resolution graph g.

`Index(v): GrphResVert -> RngIntElt`

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: GrphResVert, GrphResVert -> BoolElt`

Returns `true` if and only if the resolution graph vertices v and w are the same object in Magma.

125.2.2 Creation from Curve Singularities

Let C be a reduced plane curve, either affine or projective. See Chapter *Algebraic Curves* 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 $f\colon S \to P^2$ 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): Sch, Pt -> GrphRes`

```
M: RngIntElt                    Default: 1
K: RngIntElt                    Default: 1
```

`ResolutionGraph(C, p): Sch, Pt -> GrphRes`

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: Resolution Graph of a Curve Singularity (ex-6d6d7f)

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.

125.2.3 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 *Schemes* 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): LinearSys -> GrphRes`

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): LinearSys, RngElt, RngElt -> GrphRes`

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: Resolution Graph of a Pencil (ex-5e0089)

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.

125.2.4 Creation by Hand

A resolution graph can be created by hand. This can be fiddly if the underlying graph is complicated. See Chapter Graphs for details on how to create a directed graph.

`MakeResolutionGraph(g, s, t): GrphDir, SeqEnum, SeqEnum -> GrphRes`

`MakeResolutionGraph(g, s): GrphDir, SeqEnum -> GrphRes`

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): NwtnPgon -> GrphRes`

The resolution graph corresponding to the Newton polygon N.

`UnderlyingGraph(g): GrphRes -> GrphDir`

The underlying directed graph of the resolution graph g.

125.2.5 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): GrphResVert, GrphResVert -> GrphRes`

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): GrphRes`

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): GrphResVert, GrphResVert -> GrphRes`

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): GrphResVert -> GrphRes`

The connected component of the resolution graph containing vertex v.

`CalculateMultiplicities(~g): GrphRes`

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): GrphRes`

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): GrphResVert, RngIntElt`

Change the selfintersection at vertex v of a resolution graph to n.

`ModifyTransverseIntersection(~v, n): GrphResVert, RngIntElt`

Change the number of transverse intersections at vertex v of a resolution graph to n.

125.2.6 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): GrphRes -> RngIntElt`

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): GrphRes -> SeqEnum`

The selfintersections of the vertices of the resolution graph g.

`Multiplicities(g): GrphRes -> SeqEnum`

The multiplicities of the vertices of the resolution graph g in some divisor.

`CanonicalClass(g): GrphRes -> SeqEnum`

The multiplicities of the vertices of the resolution graph g in a local representative of the canonical class.

`TransverseIntersections(g): GrphRes -> SeqEnum`

The number of transverse intersections of some curve (usually used to create g) with the vertices of the resolution graph g.

`GenusContribution(g): GrphRes -> RngIntElt`

The contribution to the genus of a plane curve of a singularity having g as its resolution graph.

`CartanMatrix(g): GrphRes -> Mtrx`

The incidence matrix of the (undirected) graph underlying the resolution graph g with selfintersections on the diagonal.

`Determinant(g): GrphRes -> RngElt`

The determinant of the Cartan matrix of the resolution graph g.

125.3 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.

125.3.1 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): Sch, Pt -> GrphSpl`

The splice diagram of the plane curve singularity of C at the point p.

`RegularSpliceDiagram(P): LinearSys -> GrphSpl`

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): GrphDir, SeqEnum, SeqEnum -> GrphSpl`

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): SeqEnum, SeqEnum, SeqEnum -> GrphSpl`

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): GrphSpl, RngIntElt -> GrphSplVert`

The vertex of the splice diagram s having index i.

`SpliceDiagram(v): GrphSplVert -> GrphSpl`

The splice diagram containing the vertex v.

`UnderlyingGraph(s): GrphSpl -> GrphDir`

The underlying directed graph of the splice diagram s.

`UnderlyingVertex(v): GrphSplVert -> GrphVert`

`Vertex(v): GrphSplVert -> GrphVert`

The vertex of the underlying graph corresponding to the vertex v of a splice diagram.

`Vertices(s): GrphSpl -> SeqEnum`

The vertices of the splice diagram s.

`RootVertex(s): GrphSpl -> GrphSplVert`

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): GrphSplVert -> RngIntElt`

The index of the vertex v of a splice diagram.

`s eq t: GrphSpl, GrphSpl -> BoolElt`

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

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.

125.3.2 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): GrphSpl -> SeqEnum`

The integer labels on the edges of the splice diagram s.

`VertexLabels(s): GrphSpl -> SeqEnum`

The integer labels on the vertices of the splice diagram s.

`TotalLinking(v): GrphSplVert -> RngIntElt`

The total linking number of the vertex v of a splice diagram.

`LinkingNumbers(s): GrphSpl -> SeqEnum`

The total linking numbers of the vertices of the splice diagram s.

`Linking(u, v): GrphSplVert, GrphSplVert -> RngIntElt`

The linking number of vertices u and v of a splice diagram.

`EdgeDeterminant(u, v): GrphSplVert, GrphSplVert -> RngIntElt`

The edge determinant of the edge joining the vertex u to the vertex v of a splice diagram.

`Valency(v): GrphSplVert -> RngIntElt`

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): GrphSpl -> BoolElt`

Returns `true` if and only if the splice diagram s is regular.

`IsReduced(s): GrphSpl -> BoolElt`

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): GrphSpl -> BoolElt`

Returns `true` if and only if the splice diagram s has a vertex with zero linking number.

`Degree(s): GrphSpl -> RngIntElt`

The linking number of the first vertex of the splice diagram s.

`EulerCharacteristic(s): GrphSpl -> RngIntElt`

The Euler characteristic of the splice diagram s.

`Size(s): GrphRes -> RngIntElt`

The number of vertices of the splice diagram s.

`Arrows(s): GrphRes -> SeqEnum`

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): GrphSplVert, GrphSplVert -> SeqEnum, SeqEnum`

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.

125.4 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`.

125.4.1 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): GrphRes -> GrphSpl`

```
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): GrphRes, GrphResVert -> GrphSpl`

The splice diagram of the resolution graph g with the condition that the vertex v will not be removed by a reduction.

125.5 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. ISBN 0-691-08380-0.
- [Neu99] W.D. Neumann. Irregular links at infinity of complex affine plane curves. *Quart. J. Math. Ox. Ser. (2)*, 50:301–320, 1999.

Chapter 126

Algebraic Surfaces

126.1 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 surfaces with serious singularities and that for ordinary projective surfaces in arbitrary dimensional ambients that are "almost non-singular". The former relies heavily on the newer desingularisation-by-blowup code (V2.21), which applies to surfaces with a zero-dimensional singular locus, and the older formal desingularisation package of Tobias Beck, which can handle arbitrary singularities but is restricted to surfaces in ordinary projective 3-space of characteristic 0. The latter relies more on Magma's coherent sheaf package. Here, almost non-singular means that only simple (A-D-E) singularities are allowed. These are terminal singularities which don't affect computations 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 *Schemes*) and coherent sheaves (see Chapter *Coherent Sheaves*) 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 most of the intrinsics. The last subsection, however, covers the resolution of singularities by local blow-ups and the functionality specifically associated with that.

The next section deals with surfaces in $\mathbf{P}^3$ with no singularity assumptions. Here we give a full description of the formal desingularisation routines and functions which work with either desingularisation method to compute birational invariants of desingularisations, 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.

126.2 Generalities

For the convenience of the user, in this section, we recall some basic properties of general Magma schemes that will often be required when creating and working with objects of the `Srfc` subtype.

126.2.1 Ambients

In Magma, it is usually necessary for the user to begin by explicitly creating an underlying *ambient* space(s) in which the scheme(s) will live. These are fully described in the general Scheme Chapter *Schemes*. The commonest are affine and ordinary projective spaces. There are also weighted and product projective spaces and the much wider range of toric ambients coming from Magma's Toric Geometry package. However, although there is a rich selection of combinatorial functionality available for toric varieties, the *scheme* functionality for general toric ambients is still rather limited compared to the older ambients. Additionally, for surfaces, much of the off-the-shelf functionality currently only applies to those which lie in an ordinary projective ambient (we will often refer to these as ordinary projective surfaces). Unlike other general scheme sub-types, e.g. curves, we have the added restriction that surfaces can only be defined over a field. This is because virtually all of the specialised geometric functionality for surfaces only works over fields. Thus the ambient space must also be defined over a field. The surface is created by specifying the polynomials in the ambient coordinates that define it as a subvariety of the ambient. For an ordinary projective surface, all defining polynomials must be homogeneous. The ambient coordinates can be labelled in the usual way by using diamond brackets. For example, to create the hypersurface with equation $x^2 + y^2 + z^2 + t^2 = 0$ in 3-dimensional projective space ($\mathbf{P}^3$) over the rational numbers

```
> k := Rationals();
> P3<x,y,z,t> := ProjectiveSpace(k,3);
> S := Surface(P3,x^2+y^2+z^2+t^2);
> S;
Surface over Rational Field defined by
x^2 + y^2 + z^2 + t^2
```

As can be seen in this example, once labelled in the command creating the ambient (`P3` here), the labels for the ambient variables `x, y, ...` can be used directly to create polynomials in the coordinate ring of the ambient. Alternatively, we can use `P3.1`, `P3.2` etc., which are synonymous with `CoordinateRing(P3).1`, ... (or even `S.1`,.. once the surface has been created) for these variables.

126.2.2 Surfaces

An algebraic surface S in Magma — an object of the `Sch` subtype `Srfc` — is an algebraic variety of dimension 2 defined over a field k . In simple terms, it is the (algebraic) set of points defined by the vanishing of a set of polynomials in an ambient space with some additional restrictions. This algebraic set has to be *irreducible*, meaning that it can't be decomposed into a union of two such proper algebraic sets. It also has to be *reduced*, meaning that the ideal generated by the defining polynomials is radical: it contains *all* of the polynomials that vanish on the set (this is not quite strictly true for projective ambients. The ideal may not be "saturated" and have primary components belonging to the redundant prime ideals of the ambient. See Section *Constructing Schemes* for more details on saturation). A scheme is called *integral* if it is both reduced and irreducible. This is because the combination of the two conditions is equivalent to the condition that the coordinate ring of every open affine subscheme is an integral domain. For the projective ambients that Magma uses, it is also equivalent to the projective coordinate ring being an integral domain or to the saturated defining ideal being prime. In fact, the condition we require that S is a variety is the stronger condition that S is *geometrically integral*. This means that when we consider the scheme defined by the same polynomials over the algebraic closure of k, it is still reduced and irreducible. If k is a perfect field, then reduced implies geometrically reduced. However, it is not so easy to test for geometric irreducibility and we currently do not do so, checking only integrality.

Below is a typical example where geometric irreducibility fails. It is a degree 4 hypersurface in affine 3-space that is irreducible over the rational field but splits into the union of two conjugate quadric hypersurfaces (which are geometrically irreducible) over the quadratic field obtained by adjoining the cube roots of unity.

```
> A<x,y,z> := AffineSpace(Rationals(),3);
> S := Scheme(A,x^4+y^4+z^4-x^2*y^2-x^2*z^2-y^2*z^2);
> S;
Scheme over Rational Field defined by
```

(continues on next page)

(continued from previous page)

```
x^4 - x^2*y^2 - x^2*z^2 + y^4 - y^2*z^2 + z^4
> IsReduced(S);
true
> IsIrreducible(S);
true
> K<a> := QuadraticField(-3);
> A<x,y,z> := AffineSpace(K,3);
> S := Scheme(A,x^4+y^4+z^4-x^2*y^2-x^2*z^2-y^2*z^2);
> IsReduced(S);
true
> IsIrreducible(S);
false
> IrreducibleComponents(S);
[
    Scheme over K defined by
    x^2 + 1/2*(-a - 1)*y^2 + 1/2*(a - 1)*z^2,
    Scheme over K defined by
    x^2 + 1/2*(a - 1)*y^2 + 1/2*(-a - 1)*z^2
]
```

The main reason for requiring surfaces to be varieties is that much of the surface functionality involves classification and invariant intrinsics that only make sense for varieties. Specialised families like anticanonically-embedded Del Pezzos are all families of varieties.

126.2.3 Singularity

One of the most important attributes of a Magma surface is its degree of singularity. Ideally, we would like to be able to work with a surface model that has arbitrary singularities and to be able to deduce information about the projective non-singular surfaces in its birational equivalence class. Magma is able to do this for s curves by computing a full projective desingularisation behind the scenes using the function field machinery. Unfortunately, the situation for surfaces is much more complicated and we have limited resources for working with singular models at the moment, though that has now improved considerably with the introduction of code for the desingularisation of surfaces by local blow-up (V2.21). This still has the restriction (currently), however, that the singular locus of the surface model is zero-dimensional: there cannot be curves in the surface S along which S is everywhere singular. There is also an older package for computing *formal* desingularisations of surfaces in ordinary projective 3-spaces (hypersurfaces) which can handle arbitrary singularities. However, this can be very slow and inefficient and is not suitable for automatic linkage in the background. In addition, if we start with an ordinary projective surface with mild singularities in a higher-dimensional space, projection to a hypersurface can introduce much worse singularities which can be very difficult to resolve (The same is partly true for curves, where construction of the function field implicitly involves a projection. However, the desingularisation process is much more efficient there). For surfaces with curve singularities, in ambients of dimension > 3, it is probably better for the user to try to re-embed them (in a slightly higher-dimensional ambient, possibly) to give a model with only point singularities than to project to $\mathbf{P}^3$. Having said that, the blow-up desingularisation seems to work quite well in 3 and 4 dimensional ambients for 'nice' surface models but is probably much slower in general in the higher-dimensional ones.

In the curve case, affine models can be used even when calling intrinsics that compute global properties of projective curves. This is because a projective closure can be computed and the singularities at infinity of that resolved (as well as the finite singularities). That additional singularities at infinity of a particular projective closure of an affine surface cannot be automatically resolved is one of the reasons that many of the surface intrinsics that compute global properties of projective surfaces must take a projective surface as the argument.

Many of the general surface intrinsics currently assume either non-singularity or something slightly weaker (only simple/ADE/du Val singularities). Since checking singularity can be a very time-consuming process for surfaces in higher-dimensional ambients (and actually take longer than the main computation), most of the intrinsics that rely on singularity assumptions do not perform the singularity check by default,

although there is a parameter that can be set to `true` to force the check. It is important, though, that the user is familiar with the Magma intrinsics to test for different types of global/local singularity so he or she may apply them directly if desired.

There are general scheme intrinsics for testing local/global (non)-singularity in the usual sense of smoothness as well as for a scheme being Gorenstein, Cohen-Macaulay, or the arithmetic versions of these. They are described in Section *Global Geometry of Schemes* and Section *Local Geometry of Schemes* of the `Scheme` chapter. There are also intrinsics to test for normality and simple singularities which can be found in the later Section *Singularity Properties* of this chapter.

126.2.4 Maps and Points

Useful scheme features include the ability to create rational maps between schemes and to create points on the scheme defined over the base field or over an extension field. The user should familiarise himself with the system for points and pointsets described in detail in Section *Rational Points and Point Sets* and the system for maps in Section *Maps between Schemes*. Briefly, all maps are rational maps, meaning that they may not be everywhere defined, but should be defined on a Zariski-dense (i.e. not contained in a proper closed subscheme) open subscheme of the domain. They are given by a sequence of the correct length of polynomials or rational functions in the domain variables. This sequence must have the right overall homogeneity to match the ambient gradings if the domain or codomain is not affine. There is a base scheme of the map that is the set of points where the defining polynomials do not give a legitimate point of the codomain (e.g. if all polynomials evaluate to zero and the codomain is in ordinary projective space). This base scheme can be changed by adding alternative defining polynomials that are compatible (in map terms) with the original set and may extend the open subscheme where the map is legitimately defined. There is also an alternative type of map between ordinary projective schemes - a *graph map* - where the map is defined by its graph as in classical algebraic geometry. A useful intrinsic is `Extend` which will add sets of alternative equations that cover the maximal possible open set where the map can be defined as a rational map. This can be computationally quite heavy though, and the graph map alternative is often better for this, automatically giving a maximal domain of definition by a single saturation computation. For birational maps, it is also possible to add inverse defining polynomials and there are intrinsics to check for (birational) invertibility and computation of inverse equations.

Points on a scheme can be specified in the natural way, by giving their coordinates with respect to the ambient variables. For projective ambients, equivalent sets of coordinates can represent the same point, the equivalence depending on the gradings. For example, in ordinary projective space, the coordinates are the standard homogeneous coordinates and two sequences of coordinates give the same point if and only if the second sequence is equal to the first multiplied by a non-zero scalar. The image of a point under a map can be computed for any point that does not lie in the base scheme of the map. More generally, the user can get the images and inverse images of subschemes of the domain/codomain. Here is a simple example of a map between two surfaces in projective 3-space, a point, and its image under the map.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Surface(P,x^4+y^4-z^4-t^4);
> Y := Surface(P,x^2+y^2-z^2-t^2);
> f := map<X->Y|[x^2,y^2,z^2,t^2]>;
> IsEmpty(BaseScheme(f)); // is the map defined at all points?
true
> p := X![1,1,-1,-1]; // point on X
> p;
(-1 : -1 : 1 : 1)
> f(p); //image of p under f
(1 : 1 : 1 : 1)
```

126.2.5 Sheaves and Divisors

The packages for coherent sheaves and divisors contain more advanced functionality that will often be useful for work with surfaces. Some of the intrinsics there are surface specific (intersection numbers, for example) although most apply more generally. Many of the intrinsics for general surfaces to be described later in this chapter also make use of these packages. The user is urged to consult the relevant sections of the documentation for detailed information: the `Coherent Sheaves` chapter for sheaves and the Section *Divisors* of the `Scheme` chapter for divisors.

126.3 General Surfaces

126.3.1 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. The latter restriction is a consequence of the reliance on the coherent sheaf machinery for some key operations. 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, singularity checks are usually *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 $\mathbf{P}^4$.

126.3.2 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, is 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).

See Section *Creation of General Del Pezzos* for some specific creation intrinsics for Del Pezzo surfaces and Section *Parametrization of Del Pezzo Surfaces* for some additional degree 6 Del Pezzo constructors. Additionally, see Section *Special Surfaces in Projective 4-space* for intrinsics to create a range of surfaces in $\mathbf{P}^4$ belonging to special families.

`Surface(A, I): Sch, ModMPol -> Srfc`

`Surface(A, f): Sch, RngMPolElt -> Srfc`

`Surface(A, S): Sch, SeqEnum -> Srfc`

```
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 *Constructing Schemes*), `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 (reduced and irreducible) 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 stated above, our actual requirement is that surfaces are geometrically integral (equivalently, the surface is integral and 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.

`RationalRuledSurface(P, n): Prj, RngIntElt -> Srfc, MapSch`

Returns a rational, ruled surface X in the ordinary, projective ambient $P = \mathbf{P}^m$ with parameters $n, m-1-n$ where n is the second argument. Such a surface is a rational scroll that can be defined in a number of equivalent ways (see Appendix A2H, [Eis05]).

Let $\mathbf{P}^n$ and $\mathbf{P}^{m-1-n}$ be the linear subspaces of P corresponding to the first $n+1$ coordinates and the last $m-n$ coordinates respectively. Then X is given geometrically as the union of the lines L_{QR} joining a point Q on a rational normal curve in $\mathbf{P}^n$ to a point R on a rational normal curve in $\mathbf{P}^{m-1-n}$, where Q and R correspond under a fixed isomorphism of the first rational normal curve to the second. In the cases $n = 0$ and $n = m-1$, the first or second rational normal curve degenerates to a single point and X is the cone of all lines from a rational normal curve in a hyperplane of P to a point (the apex) outside of the hyperplane. The apex is the only singular point of the surface X (and is not a simple singularity in general). In the non-degenerate cases, X is non-singular. n must always be between 0 and $m-1$ (inclusive).

Following the notation of Section 2, Chapter 5 of [Har77], the rational ruled surface with parameters r, s can also be defined as follows. If e is $\mathrm{Max}(r,s)$-$\mathrm{Min}(r,s)$ and v is $\mathrm{Max}(r,s)$, then X is the Hirzebruch surface X_e (Thm. 2.17, *ibid*) mapped into P via the linear system $|C_0 + v * f|$, which gives an embedding precisely in the non-degenerate cases.

The second return value is a scheme map f from X to $\mathbf{P}^1$ which defines the ruling on X: the fibres of f are all lines in P.

`RandomCompleteIntersection(P, ds): Prj, SeqEnum[RngIntElt] -> Sch`

```
Nonsingular: BoolElt                    Default: true
RndP       : RngIntElt                  Default: 1
```

This is the same as the general scheme intrinsic to generate a random complete intersection scheme in ordinary projective space $P = \mathbf{P}^m$ over a finite field or the rationals. The variable ds should be a sequence of positive integers of length $m-2$. The intrinsic will generate random homogeneous polynomials $F_1, \ldots, F_{m-2}$ of degrees $ds[1], \ldots, ds[m-2]$ in the coordinate ring of P and return the subscheme X of P with the F_i as defining equations. It is checked that X has dimension 2 (in which case all irreducible components have dimension 2). If parameter `Nonsingular` is set to the default value of `true`, the non-singularity (actually smoothness) is also checked. This guarantees that X is geometrically integral and the result is returned as a surface type `Srfc`. If the check is not performed, X is constructed as a plain `Sch`. If X fails the dimension or non-singularity check, a new set of random polynomials is generated.

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 $+RndP$ inclusive. The default value here is 1.

`KummerSurfaceScheme(C): CrvHyp -> Srfc`

Returns the Kummer surface of the Jacobian J of the genus 2 hyperelliptic curve C. This is a singular model of the surface: a quartic hypersurface in $\mathbf{P}^3$ with 16 simple "A1" singularities corresponding to the 16 points of order 1 or 2 on J. Its desingularisation is a K3 surface.

Example: Creating Surfaces (ex-7833c6)

We illustrate the basic creation functions with some simple examples. Firstly, we create a degree 3 (Del Pezzo) and a degree 4 (K3) surface in $\mathbf{P}^3$ directly by giving a defining equation.

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Surface(P3,x^3+y^3+z^3+t^3);
> X;
Surface over Rational Field defined by
x^3 + y^3 + z^3 + t^3
> X := Surface(P3,x^4+y^4+z^4+t^4);
> X;
Surface over Rational Field defined by
x^4 + y^4 + z^4 + t^4
```

A degree 5 Del Pezzo surface in $\mathbf{P}^5$ can be created by specifying it as the projective plane blown up in 4 points (and anti-canonically embedded).

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> pts := [* P2![1,0,0],P2![0,1,0],P2![0,0,1],P2![1,1,1] *];
> X := DelPezzoSurface(P2,pts);
> P5<x,y,z,s,t,u> := Ambient(X);
> X;
Del Pezzo Surface of degree 5 over Rational Field defined by
-y*z + x*s + s^2 - s*t - s*u + t*u,
-y*s + s^2 + x*t - s*t,
-z*s + s^2 + x*u - s*u,
-s^2 + s*t + y*u - t*u,
-s^2 + z*t + s*u - t*u
```

The next example is a random surface in $\mathbf{P}^5$ which is the complete intersection of hypersurfaces of degrees 2, 2, and 3

```
> P5<x,y,z,s,t,u> := ProjectiveSpace(Rationals(),5);
> X := RandomCompleteIntersection(P5,[2,2,3]);
> X;
Surface over Rational Field defined by
-y*z-z^2-x*s+y*s+z*s-s^2-x*t-y*t-z*t+s*t+t^2-x*u-y*u-z*u-t*u,
-x^2+z^2-y*s+s^2+x*t+y*t-z*t+s*t+z*u-s*u+t*u-u^2,
-x^3-x^2*y+y^2*z-x*z^2+y*z^2-x^2*s+x*z*s+y*z*s+z^2*s+x*s^2-y*s^2+z*s^2+s^3-x^2*t+
x*z*t-z^2*t-x*s*t-x*t^2+y*t^2+z*t^2+s*t^2-t^3+x*y*u-y^2*u+x*z*u+y*z*u+y*s*u+z*s*u
-s^2*u-x*t*u+y*t*u-z*t*u+s*t*u+t^2*u+x*u^2-y*u^2+t*u^2-u^3
```

The next example is a rational ruled surface in $\mathbf{P}^4$ with parameters $2, 1$. This is a nonsingular surface scroll that is abstractly isomorphic to the Hirzebruch surface X_1 (the plane blown up at one point).

```
> P4<x,y,z,s,t> := ProjectiveSpace(Rationals(),4);
> X := RationalRuledSurface(P4,2);
> X;
Surface over Rational Field defined by
-z*s + y*t,
-y*s + x*t,
-y^2 + x*z
```

Finally, we call one of the intrinsics from the section on surfaces in $\mathbf{P}^4$ to get an Abelian surface (2-dimensional Abelian variety) which is the zero locus of a random global section of the famous Horrocks-Mumford vector bundle. There are 18 defining equations of degrees 5 and 6 that we do not list.

```
> P4<x,y,z,s,t> := ProjectiveSpace(Rationals(),4);
> X := RandomAbelianSurface_d10g6(P4);
> #DefiningPolynomials(X);
```

(continues on next page)

(continued from previous page)

```
18
> [TotalDegree(f) : f in DefiningPolynomials(X)];
[ 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6 ]
```

126.3.3 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 *Adjoint Systems and Birational Invariants*. Key invariants are stored when computed.

GeometricGenus(S): Srfc -> RngIntElt

```
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): Srfc, RngIntElt -> RngIntElt

```
CheckGor: BoolElt                    Default: false
```

The surface S and parameter **CheckGor** are the same for the **GeometricGenus** intrinsic above. The integer n should be non-negative. Returns the dimension of the space of global sections of the n-th tensor power of the canonical sheaf K of S. Again this will generally be larger than the n-th plurigenus of a desingularization of S unless S has at worst simple singularities.

ArithmeticGenus(S): Srfc -> RngIntElt

Given a scheme S, this function returns the arithmetic genus. It is, in fact, the general scheme intrinsic.

Irregularity(S): Srfc -> RngIntElt

```
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 they have already been established. See next section.).

If **CheckGor** is set to **true** (the default is **false**), and the above conditions are not satisfied, the Gorenstein condition 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): Srfc, RngIntElt -> RngIntElt`

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 n-th 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): Srfc, RngIntElt -> RngIntElt`

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): Srfc, RngIntElt, RngIntElt -> RngIntElt`

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: Invariants of a Kummer Surface `(ex-a227cc)`

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 and 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.

```
> f := PolynomialRing(Rationals())![-1,0,0,0,0,0,1]; //t^6-1
> X := KummerSurfaceScheme(HyperellipticCurve(f));
> IsSingular(X);
true
> HasOnlySimpleSingularities(X);
true
```

(continues on next page)

(continued from previous page)

```
> 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
```

126.3.4 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 *Schemes* 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): Srfc -> BoolElt`

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): Pt -> BoolElt, MonStr, RngIntElt`

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. Therefore in characteristic 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 iso-

morphic 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., that 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): Srfc -> BoolElt, List`

```
ReturnList: BoolElt                    Default: false
```

This intrinsic determines whether the surface S has no singularities worse than isolated simple singularities as described in the `IsSimpleSurfaceSingularity` 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 recorded internally) that there are only simple singularities without their precise type having been computed. For example, if S is a minimal or weighted canonical model of a surface of general type.

Example: Simple Singularities of a Degenerate Del Pezzo (ex-d1e251)

Anticanonically-embedded *degenerate* Del Pezzo surfaces of degree ≥ 3 are singular but have only simple singularities. This is verified for a degree 4 Del Pezzo which has two 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
]
```

As a second example, we consider a singular rational ruled surface (scroll) that is the cone over a rational normal curve. The intrinsics tell us that the surface is normal and Cohen-Macaulay (i.e., the local ring at the singular point at the apex of the cone satisfies these properties) but that it satisfies none of the stronger "non-singularity" properties.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> X := RationalRuledSurface(P4,0);
> // one singular point
> Degree(ReducedSubscheme(SingularSubscheme(X)));
1
> Support(SingularSubscheme(X));
{ (1 : 0 : 0 : 0 : 0) }
> HasOnlySimpleSingularities(X);
false
> IsArithmeticallyGorenstein(X);
false
> IsGorenstein(X);
false
> IsCohenMacaulay(X);
true
```

(continues on next page)

(continued from previous page)

```
> IsNormal(X);
true
```

126.3.5 Kodaira-Enriques Classification

`KodairaEnriquesType(S): Srfc -> RngIntElt, RngIntElt, MonStr`

```
KnownADE: BoolElt                    Default: false
```

The argument S is a surface in ordinary projective space having at most point singularities (i.e. a singular subscheme of dimension ≤ 0), unless it lies in a $\mathbf{P}^3$ in characteristic 0, when a formal desingularisation as described in Section *Formal Desingularization of Surfaces* may be used. In the point singularity case, blow-up desingularisations as described in Section *Formal Desingularization of Surfaces* are used. In either case the desingularisations are used to compute plurigenera and plurigeneric maps. In the case where S has at most simple (A-D-E) singularities, it is generally faster to use sheaf methods for these computations, so the simple singularity condition is checked for at the start. If it is known beforehand that the singularities are at worst simple, the user may set the `KnownADE` parameter to `true` (the default is `false`) to skip this check.

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. 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.

`KodairaDimension(S): Srfc -> RngIntElt`

```
KnownADE: BoolElt                    Default: false
```

The argument S is a surface in ordinary projective space satisfying the same conditions as for the `KodairaEnriquesType` intrinsic and the `KnownADE` parameter is also the same as for that intrinsic. 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: Kodaira-Enriques Types of Three Surfaces (ex-a66628)

We will present further cases of Kodaira-Enriques typing in our minimal model examples. For now, we just give three simple examples: a Veronese surface in $\mathbf{P}^5$, the Kummer surface in $\mathbf{P}^3$ with simple singularities from our earlier example, and a non-minimal K3 surface in $\mathbf{P}^3$ with non-simple singularities.

The scheme X is a Veronese surface, isomorphic to $\mathbf{P}^2$:

```
> 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 with simple singularities:

```
> 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
```

The scheme X is a singular K3 surface with non-simple point-singularities:

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Surface(P,x^6+2*x^5*y-2*x^4*y^2+2*x^2*y^4+y^6+x^4*t^2+2*x^3*y*t^2+
>   x^2*y^2*t^2-x*y^3*t^2-2*y^4*t^2-x^2*y*t^3-2*x*y^2*t^3+2*x^2*t^4+y^2*t^4-
>   z^2*t^4-x*t^5);
> KodairaEnriquesType(X);
0 -2 K3
```

126.3.6 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 (certain Del Pezzo surfaces, for example) that are not minimal over the algebraic closure. Models like Del Pezzo surfaces that are close to minimal but may not strictly even be minimal over k are still very important for rational and ruled surfaces because they allow the reduction to a small class of standard isomorphism types. We can think of these as quasi-minimal.

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 really 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 C that presents the minimal model M as an elliptic fibration over C. In this case, M is the global arithmetic minimal model of its generic fibre.

`MinimalModelRationalSurface(S): Srfc -> Map`

`CheckSing: BoolElt` Default: `true`

Let S be a non-singular ordinary projective rational (Kodaira dimension -1 and irregularity 0) surface. Non-singularity is checked by default; set `CheckSing` to `false` to skip this check. It is also left to the user to check that S is rational (using *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: Minimal Model of a Rational Surface `(ex-c7d6da)`

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): Srfc -> Map

```
CheckSing: BoolElt                    Default: true
```

The surface S should be non-singular ordinary projective of Kodaira dimension -1. Non-singularity is checked by default; set `CheckSing` to `false` to skip this check. 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): Srfc -> Map

```
CheckSing: BoolElt                    Default: true
```

The surface S should be non-singular ordinary projective of Kodaira dimension 0. Non-singularity is checked by default; set `CheckSing` to `false` to skip this check. 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: Minimal Model of a Blown-Up Torus (ex-19dd97)

We start with an example of a non-minimal surface X 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 an X. It has a natural embedding in $\mathbf{P}^7$. This turns out to be 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.

```
> SetSeed(1);
> 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: KnownADE);
0 -1 Torus
> mp := MinimalModelKodairaDimensionZero(X);
> Y := Codomain(mp);
> Ambient(Y);
Projective Space of dimension 8 over Rational Field
```

(continues on next page)

(continued from previous page)

```
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: Minimal Model of a Blown-Up Enriques Surface (ex-4ca886)

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): Srfc -> Map, Map`

```
CheckSing: BoolElt                    Default: true
Fibration: BoolElt                    Default: false
```

Let S be a non-singular ordinary projective surface of Kodaira dimension 1. Non-singularity is checked by default; set `CheckSing` to `false` to skip this check. 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): Srfc -> Map, BoolElt

CanonicalWeightedModel(S): Srfc -> Map, BoolElt

```
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): Srfc -> RngMPol

```
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: Minimal Model of a Horikawa Surface (ex-99ef0e)

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: KnownADE);
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 -
```

(continues on next page)

(continued from previous page)

```
 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: Minimal Model of a Blown-Up Quintic Surface (ex-c59458)

This example is a non-singular but non-minimal surface. We start with a degree 5 hypersurface in $\mathbf{P}^3$ and then blow up a point using the standard intrinsic to give a model X of the blow-up as a surface in $\mathbf{P}^8$. The original hypersurface is a (simply-weighted) canonical model of X and `WeightedCanonicalModel` just returns a slight linear transformation of it. The computation takes a little time.

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> Y := Surface(P3,x^5+y^5+z^5+t^5 : Nonsingular := true); //the hypersurface
> X := Blowup(Y,Y![0,0,-1,1]);
> P<x1,x2,x3,x4,x5,x6,x7,x8,x9> := Ambient(X);
> X;
Surface over Rational Field defined by
-x6*x8+x5*x9, -x3*x8+x2*x9, -x1*x8+x3*x9, -x6*x7+x4*x9, -x5*x7+x4*x8,
-x3*x7+x5*x9+x8*x9, -x2*x7+x5*x8+x8^2, -x1*x7+x6*x9+x9^2,
-x3*x5+x2*x6, -x1*x5+x3*x6, -x3*x4+x5*x6+x6*x8, -x2*x4+x5^2+x5*x8,
-x1*x4+x6^2+x6*x9, -x1*x2+x3^2,
x1^4+x2*x3^3+x4*x6^3-x4*x6^2*x9+x4*x6*x9^2-x4*x9^3+x7*x9^3,
x1^3*x3+x2^2*x3^2+x4*x5*x6^2-x4*x5*x6*x9+x4*x5*x9^2-x4*x8*x9^2+x7*x8*x9^2,
x2*x3^2*x5+x1^3*x6+x4^2*x6^2+x2*x3^2*x8+x1^3*x9-x4^2*x6*x9+x4^2*x9^2-
x4*x7*x9^2+x7^2*x9^2,
x2^3*x3+x1^2*x3^2+x4*x5^2*x6-x4*x5^2*x9+x4*x5*x8*x9-x4*x8^2*x9+x7*x8^2*x9,
x2^2*x3*x5+x1^2*x3*x6+x4^2*x5*x6+x2^2*x3*x8+x1^2*x3*x9-x4^2*x5*x9+x4^2*x8*x9-
x4*x7*x8*x9+x7^2*x8*x9,
x2*x3*x5^2+x4^3*x6+x1^2*x6^2+2*x2*x3*x5*x8+x2*x3*x8^2-x4^3*x9+2*x1^2*x6*x9+
x4^2*x7*x9-x4*x7^2*x9+x7^3*x9+x1^2*x9^2,
x2^4+x1*x3^3+x4*x5^3-x4*x5^2*x8+x4*x5*x8^2-x4*x8^3+x7*x8^3,
x2^3*x5+x4^2*x5^2+x1*x3^2*x6+x2^3*x8-x4^2*x5*x8+x4^2*x8^2-x4*x7*x8^2+
x7^2*x8^2+x1*x3^2*x9,
x4^3*x5+x2^2*x5^2+x1*x3*x6^2-x4^3*x8+2*x2^2*x5*x8+x4^2*x7*x8-x4*x7^2*x8+
x7^3*x8+x2^2*x8^2+2*x1*x3*x6*x9+x1*x3*x9^2,
x4^4+x2*x5^3+x1*x6^3-x4^3*x7+x4^2*x7^2-x4*x7^3+x7^4+3*x2*x5^2*x8+
```

(continues on next page)

(continued from previous page)

```
3*x2*x5*x8^2+x2*x8^3+3*x1*x6^2*x9+3*x1*x6*x9^2+x1*x9^3
> KodairaEnriquesType(X: KnownADE);
2 0 General type
> mp, is_min := CanonicalWeightedModel(X);
> is_min;
false
> X1 := Codomain(mp); //the canonical model
> P<a,b,c,d> := Ambient(X1);
> X1;
Surface over Rational Field defined by
a^5 + b^5 + c^5 + 5*c^4*d + 10*c^3*d^2 + 10*c^2*d^3 + 5*c*d^4 + 2*d^5
> MinimalChernNumber(X,1) - ChernNumber(X,1);
1
```

The last line confirms that there was one exceptional divisor blown down.

126.3.7 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): Prj -> Srfc`

```
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): Prj -> Srfc`

```
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): Prj -> Srfc`

```
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): Prj -> Srfc`

```
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): Prj -> Srfc`

```
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): Prj -> Srfc`

```
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): Prj -> Srfc`

```
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.

126.4 Desingularisation by Blow Up

126.4.1 Introduction

In addition to the older functionality for formal desingularisation of hypersurfaces in $\mathbf{P}^3$ in characteristic zero described in the next section, there is now (Magma V2.21) a more general desingularisation routine

available that desingularises by blowing up. It has the current restriction that it only applies to surfaces with point singularities (the singular subscheme is of dimension zero) but this should only be a provisional restriction that is lifted in later releases.

The advantages are firstly that it applies to surfaces in any ambient of any characteristic with the mild proviso that all of the singular points have to lie in definable affine patches. In particular, it applies to surfaces in affine, projective, or product projective space of any dimension. Secondly, it provides a much more detailed description than formal desingularisation. Global information is available for each irreducible divisor lying over each singular point rather than just generic information. This allows the computation of the intersection pairing between these blow-up divisors as well as computation of intersection pairings between the blow-up divisors and strict transforms (pull-backs to the desingularisation with blow-up divisor components removed) of divisors on the singular surface model. This is in addition to the functionality for computing the multiplicity of blow-up divisors in the pull-backs of divisors. The latter should be available for formal desingularisation also but is not fully implemented currently. There is much useful information regarding the canonical divisor on the desingularisation that has been extended from hypersurfaces only to any ordinary projective surface in Magma V2.25. Finally, it has the advantage that it is often much faster than formal desingularisation in the ordinary hypersurface case where both apply. In the main resolution intrinsic *ResolveSingularSurface*, it is now used as the default method, even in the hypersurface case, as long as the surface has only point singularities. We will illustrate the speed and rich range of functionality in three extended examples at the end of this section.

Let k be the base field of the surface S. In resolving by blow-up, there is a choice to make. When blowing up a zero-dimensional reduced subscheme, we can choose to blow up the whole thing in one operation; split into irreducible components, which represent a conjugate set of points in $S(\bar{k})$, and blow these up separately; or keep base-changing to fields over which all points are rational and blow up only rational points. We have chosen the second method. Splitting into irreducible components has some overhead cost, but we cannot identify the different irreducible blow-up components at the end without it. We also do not base-change in order to blow up only rational points, because it is probably best to stick to working over the original ground field k. Groebner basis computations in Magma are very highly-tuned over the rational numbers, for example, but may not work quite so efficiently over number fields. On the other hand, blowing up an irreducible zero-dimensional subscheme rather than the points over an extension field can lead to an increase in the number of variables for the ambients that our blow-up patches lie in. However, we usually have to blow up singular curves at some stage in the resolution process and it is hard to avoid a potential increase in the number of variables in these operations, anyway. In addition, when we avoid field extensions, the final 'irreducible' blow-up components over a singular point will often not all be geometrically irreducible (never in the case of blowing up a 'non-rational singular point'), but some will split into a set of conjugate irreducible components over $\bar{k}$. This, however, should not cause many problems. For example, multiplicities of conjugate blow-up divisors in rational divisors or differentials are all the same, so they always enter into decompositions of rational divisors as sums of our blow-up components with multiplicity. If the user really needs to inspect the blow-up divisors in detail, rather than just using the top-level intrinsics, there are functions to return affine patches of them and they can be used to check for geometric irreducibility or not. All of the package intrinsics only require irreducible over k blow-up components for their implementation.

ResolveSingularSurface(S): Srfc -> List, RngIntElt

```
AdjComp         : BoolElt                    Default: false
UseFormalDesing: BoolElt                    Default: false
Resolve         : Verbose                   Default: 1
```

The main intrinsic to compute a desingularisation of surface S and return a data list containing information about it. The desingularisation is either formal or of blow-up type. The second return value is a code value for the type: 1 for formal and 2 for blow-up desingularisation. The type of desingularisation is determined as follows. If the characteristic of the base field k is not 0, or S does not lie in ordinary projective space of dimension 3, $\mathbf{P}^3$, only blow-up desingularisation is possible. However, this currently requires S to have only point singularities (its singular subscheme must be empty or of dimension 0). If only blow-up type is possible and S has curves in its singular locus, a runtime error will result. When the ambient of S is $\mathbf{P}^3$, k is of characteristic 0 and the singularities are of dimension 0, either method could be used but blow-up is taken as the default. To force formal

desingularisation in the case of a choice, the parameter `UseFormalDesing` can be set to `true` (the default is `false`). If the user only wishes to perform adjoint computations (see `HomAdjoints`), determine the arithmetic, geometric, or pluri-genera of the desingularisation, or perform most other computations that only depend on the canonical divisor class of the desingularisation, some time may be saved by setting parameter `AdjComp` to `true`. This means that the resolution is not fully carried out, but the only singularities in the desingularisation process that are not fully resolved are of 'canonical type': none of the resolution curves above them occur in the support of some canonical divisor and they do not impose conditions on the sections of the canonical sheaf. For blow-up resolution, it is only simple singularities on the base singular scheme S that are ignored when `AdjComp` is true (we do not check for ignorable singularities arising at later stages) and it probably saves little time using this option in general, as simple singularities should resolve quite quickly under blow-up. However, it can save more time with formal desingularisation.

In neither case is the resolution guaranteed to be a *minimal* one. That is, there may exist (-1)-curves in the irreducible components lying above singularities.

The elements of the list returned are of type `DesingData` for blow-up resolution and tuples containing a sequence of four algebraic power series and a multiplicity for formal desingularisation. In the formal case, there is a tuple for each exceptional divisor (lying over a singular point or curve on S) of the desingularisation and the power series define the map from the completion of the local ring of the generic point of that divisor to S. This is all described more fully in the next section on surfaces in $\mathbf{P}^3$. For blow-up desingularisation, there is a `DesingData` object for each irreducible component p of the singular subscheme of S (these correspond to a conjugate set of singular points in $S(\bar{k})$ - we will usually just refer to each one as a singular point from now on, whether it is a k-rational singular point or a set of conjugates). Each object contains a set of affine patches arising in the blow-up process along with birational blow-down maps to a chosen affine patch of S containing p; a set of irreducible curves for each patch which give the intersection of the blow-up divisors with that patch; and information to determine which irreducible curves for different patches are parts of the same complete irreducible blow-up divisor over p. They also cache other useful information about the resolution over p that has been determined by post-resolution intrinsic calls.

Once a formal or blow-up resolution has been computed, it is stored in an attribute of S for use in other intrinsic calls or to be returned by subsequent calls to this intrinsic. Resolution data for both formal and blow-up decomposition will be stored if resolutions of both types are constructed at different times. If a blow-up data list is computed and stored for Adjoint computation only and a subsequent call to *ResolveSingularSurface* or equivalent is made with `AdjComp` `false`, the stored resolution data is upgraded to full resolution data by computing blow-up resolutions over the simple singularities only: the entire resolution process is not repeated.

`DesingulariseSurfaceByBlowUp(S): Srfc -> List`

```
AdjComp: BoolElt                    Default: false
Resolve: Verbose                    Default: 1
```

This is just a variation of intrinsic `ResolveSingularSurface` above that always uses blow-up desingularisation. The conditions on S are the same and the returned resolution list is again cached.

`ResolveSingByBlowUp(X, Y): Sch, Sch -> DesingData`

```
Resolve: Verbose                    Default: 1
```

This is another variant that we have exported. In this case, X must be an affine surface and Y an irreducible, reduced subscheme of dimension zero, which is a single singular 'point' (i.e. a conjugate set of points over $\bar{k}$) on X. Note that the conditions on Y are not checked here. It performs the blow-up resolution of just Y and returns the desingularisation data object for it. We have exported this so the user can have a look at the desingularisation of individual singularities in case the overall desingularisation process is a little slow. If the original surface is a projective one, it should be applied to an affine patch that contains the singularity.

126.4.2 Accessor Functions

Functions to obtain some basic information about the blow-up resolution over a singular point. These are not needed for the main Magma functionality related to desingularisation but they are provided for times when the user may wish to examine the local structure of the resolution in more detail.

`NumberOfBlowUpDivisors(dsd): DesingData -> RngIntElt`

The number of irreducible divisors (curves) lying over the surface singular point p for which *dsd* is the desingularisation data object.

`SingularPoint(dsd): DesingData -> Sch`

The singular point on a surface S for which *dsd* is the blow-up desingularisation object. Note that the 'point' is not necessarily a point of $S(k)$, where k is the base field of S. It is a reduced, irreducible (zero-dimensional) component of the singular locus of S and so is a scheme-theoretic point representing a conjugate set of singular points of $S(\bar{k})$. For this reason it is returned as a subscheme of S rather than as a point type.

`BlowUpDivisor(S, dsd, i): Srfc, DesingData, RngIntElt -> Crv, Sch, MapSch`

Given *dsd*, the desingularisation data object for the blow-up resolution of point singularity p on surface S, returns an affine curve C giving one patch of the i-th irreducible divisor lying over p, the surface blow-up affine patch Sa in which it lies, and the blow-down map of Sa to S.

`BlowUpDivisorAllPatches(dsd, i): DesingData, RngIntElt -> List, RngIntElt`

Let *dsd* be the desingularisation data object for the blow-up resolution of point singularity p on surface S. This intrinsic returns the most detailed information about the i-th irreducible divisor lying over p. It gives a full set of affine patches in the blow-up resolution of p that cover the complete divisor. As it involves a rather complicated set-up, this intrinsic is best avoided unless the user really knows what they are doing. The description and comments below should be read with care.

The first actual return value is a list of 4-tuples, one for each patch. The entries of the tuples are

(i) An affine curve C giving the irreducible divisor on the patch.

(ii) The affine surface Sa which is the blow-up patch in which C lies.

(iii) The birational blow-down map of Sa to an affine patch of S in which p lies.

(iv) The index of Sa in the full list of blow-up patches in *dsd*.

The patch index of the affine patch of S (containing p), which is the Codomain of all of the maps which are third entries of a tuple, is the second return value. If S is itself affine, this second return value is just 0.

We choose to gives maps to the base patch here, rather than to S as in the last intrinsic, because it makes it slightly more efficient to compose one with the inverse of another to get the rational map from one curve patch to another. However, it should be noted that the rational maps between patches given directly as the blow-down composed with the blow-up do not generally restrict to maximally-defined maps on the affine curves and may have the entire affine curve in their base-scheme in some cases.

Also, we note that *not all* blow-up patches of the surface are non-singular. So as not to do redundant work and to avoid multiple copies of blow-up divisors, we only blow up one copy of isolated singular points, which will generally occur in multiple blow-up affine patches (these affine patches all intersect abstractly in Zariski open subsets) that arise in the resolution process. However, over the (abstractly intersecting) affine surface patches containing the complete irreducible blow-up divisor, each point on the divisor will occur on *at least one* patch at a point where the surface is non-singular. It is on these patches (relative to the particular point) that the local description of the divisor at the point is the correct one.

The package intrinsics dealing with blow-up divisors are coded to get around these problems, so it is recommended that these be used for required operations if possible.

Example: Blow-Up Desingularisation Accessors (ex-196358)

We illustrate these basic accessors with an easy example. S is a degree 5 degenerate (weak) Del Pezzo surface in $\mathbf{P}^5$ with a simple A_1 and a simple A_2 singularity.

```
> P5<a,b,c,d,e,f> := ProjectiveSpace(Rationals(),5);
> S := Surface(P5,[b^2-a*d, b*c-a*f, c^2-b*e, c*d-b*f, d*e-c*f]);
> HasOnlySimpleSingularities(S : ReturnList := true);
true
[* <(0 : 0 : 0 : 0 : 1 : 0), "A", 2>, <(1 : 0 : 0 : 0 : 0 : 0), "A", 1> *]
> dsds := ResolveSingularSurface(S);
> #dsds; //one for each singular point
2
> [NumberOfBlowUpDivisors(dsd) : dsd in dsds];
[2,1]
> // First dsd corresponds to the A2.
> dsd := dsds[1];
> SingularPoint(dsd);
Scheme over Rational Field defined by
a, b, c, d, f
> Da,Sa,prj := BlowUpDivisor(S, dsd, 1);
> Da; Ambient(Da);
Curve over Rational Field defined by
$.1
Affine Space of dimension 2 over Rational Field
Variables: $.1, $.2
> Sa; // blow-up surface patch is just A^2!
Scheme over Rational Field defined by
0
> prj; //typical blow-up map!
Mapping from: Sch: Sa to Srfc: X5
with equations :
$.1^2*$.2
$.1^2*$.2^2
$.1*$.2
$.1^2*$.2^3
1
$.1*$.2^2
> // get patch cover of the full blow-up divisor D, which is just P^1
> lst := BlowUpDivisorAllPatches(dsd, 1);
> #lst; //2 patch cover
2
> D1 := lst[1][1]; D2 := lst[2][1];
> mp12 := Restriction(Expand(lst[1][3] * Inverse(lst[2][3])),D1,D2);
> A1<x,y> := Ambient(D1); A2<x,y> := Ambient(D2);
> // D1 and D2 are both the line x=0 in an affine plane. The rational map
> // between them equates to the usual y -> 1/y
> D1; D2; mp12;
Curve over Rational Field defined by
x
Curve over Rational Field defined by
x
Mapping from: CrvPln: D1 to CrvPln: D2
with equations :
$.1*$.2^2
1/$.2
```

126.4.3 Multiplicities, Intersections, and Restricted Linear Systems

This subsection deals with blow-up desingularisation intrinsics to do with multiplicities of irreducible blow-up divisors in the pullbacks of functions and divisors, and intersection numbers for divisors that are blow-up divisors or the strict transforms of singular surface divisors on the desingularised surface. Those involving only blow-up divisors depend only on the desingularisation object and work in any ambient. Those that involve divisors on the singular surface work for surfaces in ordinary projective space. All intrinsics use a `DesingData` desingularisation data object of blow-up type cached in singular surface S. If the resolution process hasn't already been carried out, it is performed at the start (and the result cached).

`IntersectionMatrix(dsd): DesingData -> Mtrx`

```
get_self_ints: BoolElt                    Default: true
MultsAndInts : Verbose                    Default: 1
```

Returns the intersection matrix for the irreducible (blow-up) divisors that lie above the surface singular point p whose resolution is represented by data object *dsd*. This is a symmetric matrix where the (i, j)-th entry is the intersection number $D_i.D_j$, where the D_i are the irreducible divisors. The most work involved in this is computing the self-intersections, which can be rather slow. The intersections between different divisors are generally computed very quickly. Parameter get_self_ints can be set to `false` to only compute the non-self intersections. In that case, the matrix is returned containing 0s for the diagonal entries. This should cause no confusion as all genuine self-intersections are negative numbers. The result is cached in *dsd* for future use. If the result has been cached without self-intersections and the intrinsic is called again with get_self_ints set to `true`, only the diagonal entries are computed.

`Multiplicities(S, D): Srfc, DivSchElt -> SeqEnum`

`Multiplicities(S, D): Srfc, Sch -> SeqEnum`

`Multiplicities(S, dsd, D): Srfc, DesingData, Sch -> SeqEnum`

```
MultsAndInts: Verbose                     Default: 1
```

These intrinsics give the multiplicities of the irreducible blow-up divisors above singular points in the pullback of a divisor on S to the desingularisation. Note that the usual requirement of S to be normal here for divisor computations to be correct is not needed because we are effectively working on the desingularisation.

The first two intrinsics take the singular surface S as a first argument and the divisor D as second. In the second intrinsic – for effective divisors – the divisor is passed in as a subscheme of S. The return value is a sequence of sequences: one sequence for the divisors above each of the singular points, which correspond 1-1 with the `DesingData` objects in the resolution list of S and are taken in the order of that list.

The third intrinsic returns a single sequence containing only the multiplicities above a particular singular point of S, which is represented by the `DesingData` object *dsd* passed in as second argument. The third argument is an effective divisor D given as a subscheme.

`MultiplicitiesAndIntersections(S, D): Srfc, DivSchElt -> SeqEnum, SeqEnum`

`MultiplicitiesAndIntersections(S, D): Srfc, Sch -> SeqEnum, SeqEnum`

`MultiplicitiesAndIntersections(S, dsd, D): Srfc, DesingData, Sch -> SeqEnum, SeqEnum`

```
MultsAndInts: Verbose                     Default: 1
```

The intrinsics are the exact analogues of the `Multiplicities` ones except that they also compute and return the intersection of the *strict transform* of divisor D with the irreducible blow-up divisors. The strict transform of D is the pullback of D to the desingularisation with all blow-up divisor components removed. The return values are in the same format as for `Multiplicities` except that there are now two sequences of sequences or two sequences returned. The first contains the multiplicities and the second contains the intersections with the strict transform.

This can often be much slower than `Intersections`. Most of the work usually goes into the computation of the strict transform on the blow-up surface patches, where removing blow-up divisor components can be a heavy task.

`IntersectionNumberOfStrictTransforms(S, D1, D2): Srfc, Sch, Sch -> RngIntElt`

For two effective divisors, D_1 and D_2, on (singular) surface S, given as subschemes of S, computes the intersection number $\tilde{D}_1.\tilde{D}_2$ of the strict transforms of the D_i on the blow-up desingularisation of S. Currently, S must lie in ordinary projective space and have at worst point singularities.

`SelfIntersectionOfStrictTransform(S, D): Srfc, Sch -> RngIntElt`

For an effective divisor, D, on (singular) surface S, given as a subscheme of S, computes the self-intersection number $\tilde{D}.\tilde{D}$ of the strict transform of D on the blow-up desingularisation of S. Currently, S must lie in ordinary projective space and have at worst point singularities.

`LinearSystemDivisorRestriction(S, B, ms): Srfc, SeqEnum[RngMPolElt], SeqEnum[SeqEnum] -> SeqEnum`

`CheckB: BoolElt`	`Default: true`

The surface S is a surface with only point singularities and B is a sequence of polynomials on the ambient of S which are homogeneous of the same multi-degree with respect to all gradings of S and can be thought of as sections (not necessarily linearly-independent) of a linear system on S given by the vector space V of appropriately-graded polynomials generated by B.

The sequence ms should be a sequence of sequences of integers giving some multiplicities for the irreducible blow-up divisors. This should be in the same format (in terms of grouping and order of divisors) as the return value of `Multiplicities` above.

The intrinsic computes the sublinear system of V consisting of sections whose pull-back to the blow-up desingularisation vanish on each blow-up divisor to at least the multiplicity given in ms. Note that negative and zero multiplicities have no effect. The return value is a basis of sections of this sublinear system, which will be linearly-independent modulo the defining ideal of S.

The implementation initially checks B for linear-independence modulo the ideal of S and, if necessary, replaces it with a linearly-independent subsequence that generates the same V. To save a little time, the user can set parameter `CheckB` to `false` when it is known that B gives linearly-independent sections and then this initial check is skipped.

One intended use of this intrinsic is to compute Riemann–Roch spaces of strict transforms of divisors on S. The second and third long examples at the end of this section give illustrations of this. We briefly indicate how it works generally, with D a divisor on S and $\tilde{D}$, its strict transform.

Assume that the Riemann–Roch space of D has been computed, using the sheaf or divisor functionality for example, on S. Let it be given by a basis of polynomials B_1 and denominator d (all polynomials having the same grading). Then the divisor of vanishing of d on S will be $D + R$ for some effective divisor R and B_1 is the basis of polynomials (of the same grading) whose divisor of zeros contains R. When we pull-back to the desingularisation of S, the divisor of d pulls back to $\tilde{D} + \tilde{R} + \sum_i a_i D_i$ where D_i runs over the irreducible blow-up divisors and a_i is the multiplicity of D_i in the pull back of the divisor of d. It is easy to see that the Riemann–Roch space of $\tilde{D}$ is the space of rational functions on S with basis f/d where f runs through a basis of the sublinear system of the one generated by B_1 that consists of sections that vanish on each D_i to multiplicity at least a_i, i.e. the multiplicity of d.

Warning: It is important to remember that if D_1 lies in the rational equivalence class of D on S, its strict transform will not generally be rationally equivalent to $\tilde{D}$ on the desingularisation because the two will differ by a non-principal divisor supported on blow-up divisors. So, it is important to make sure that the call to get the Riemann–Roch space on S is applied to the correct divisor. This will be seen in the third example below, blowing down lines on a degenerate Del Pezzo surface. There, a hyperplane divisor has to be added to the sum of the divisors of two lines. The hyperplane divisor has to avoid the singular points for the strict transform of the resulting divisor to be in the class that we want on the desingularised Del Pezzo.

126.4.4 Canonical Divisor Functionality

This subsection deals with blow-up resolution intrinsics that involves canonical divisors on the desingularisation. They are available for surfaces in ordinary projective space $\mathbf{P}^n$ of any dimension (as of Magma V2.25), with a few specific to $\mathbf{P}^3$. Along with the intrinsics listed here, there are also important intrinsics for birational invariants (*GeometricGenusOfDesingularization*, *ArithmeticGenusOfDesingularization* , and *PlurigenusOfDesingularization*) and for adjoint maps (*HomAdjoints*), which include pluri-canonical and pluri-anticanonical maps, that are available for resolutions of singularities of blow-up *or* formal type and are described in subsection *Adjoint Systems and Birational Invariants* of the next section on hypersurfaces.

`DifferentialMultiplicities(dsd): DesingData -> SeqEnum`

This is the fundamental intrinsic to get information about divisor multiplicities of irreducible blow-up divisors of a blow-up desingularisation in the divisors of canonical differential forms (sections of the canonical sheaf: differentials of type (2,0)) when argument `dsd` is a desingularisation data object for a *hypersurface* S. Let F be the homogeneous polynomial of degree d in 4 variables defining the singular surface S in $\mathbf{P}^3$.

Let ω be the 'natural' pseudo-differential form which is given by $dx \wedge dy/(\partial f/\partial z) = dy \wedge dz/(\partial f/\partial x) = dz \wedge dx/(\partial f/\partial y)$ on a standard affine patch of S with affine coordinates x, y, z where f is the defining polynomial of S on the patch: the dehomogenisation of F at the variable which is set to 1. With an appropriate choice of signs between the standard patches, $\pm$ this form multiplied by a homogeneous polynomial G of degree $d-4$ gives a genuine global differential form on S, ω_G. That is, the $\pm\omega$ form on the standard patches multiplied by the dehomogenisation of G on those patches gives a set of patch forms that agree on the patch intersections. On each patch, ω gives a form that has no zeros or poles outside of the singular points.

The intrinsic returns the sequence of multiplicities of the irreducible blow-up divisors which lie above the singular point p in the divisor of the pull-back of ω. Here p is the singular point represented by desingularisation object *dsd*.

The pull-back of genuine differential form ω_G has divisor equal to the (pull-back of the) effective divisor of G added to the divisor supported on all blow-up components (over all singular points) with multiplicities given by the return values of this intrinsic for the various *dsd*s.

Note that all of these multiplicities are ≤ 0 and that they all equal 0 over a singular point p if and only if p is a simple singularity. This is why simple singularities can be ignored for most canonical divisor questions.

Examples of the use of `DifferentialMultiplicities` are given in the first two long examples at the end of the section. It is also used by many of the other package intrinsics relating to the canonical divisor.

`BasisOfHolomorphicTwoForms(S): Srfc -> SeqEnum[RngMPolElt], RngMPolElt, SeqEnum, List`

`UseProj: BoolElt`	`Default: true`

Computes a basis for everywhere defined differential 2-forms [elements of $\wedge^2\Omega_1(S)$] on the desingularisation of surface S, that lies in ordinary projective space.

The third return value is a sequence $[f_1, f_2, f_3]$ representing a base meromorphic 2-form w. Here the f_i are rational functions on S given as quotients of homogeneous polynomials on the ambient of S of the same degree and f_3 is usually 1. $w = f_3 * (d(f_1) \wedge d(f_2))$.

The first 2 return values are a sequence of homogeneous polynomials $[b_1, \ldots, b_n]$ and a homogeneous polynomial F (all in the coordinate ring of the ambient of S), all of the same degree d, such that a basis of holomorphic two-forms is given by the 2-forms $(b_i/F) * w$. If the space of holomorphic two forms only contains 0, B will be the empty sequence.

The fourth return value is a data list giving the divisor of w on the desingularisation of S. This can be used in more general computations for $S \subset \mathbf{P}^n$, $n \geq 4$ in the same way that `DifferentialMultiplicities` is used for hypersurfaces in $\mathbf{P}^3$.

If S lies in $\mathbf{P}^n$, $n \geq 6$, then the intrinsic tries to simplify the differential computations by first projecting the surface isomorphically down to one in $\mathbf{P}^5$, performing the computation of the divisor of a meromorphic 2-form there, and pulling back the result. This may actually be slower, though, as the equations for the surface may become messier. To avoid this projection method, the `UseProj` parameter can be set to `false`. Its default is `true`.

Once computed, the data for the base 2-form is cached for use in other intrinsics.

`PluriCanonicalBasis(S, m): Srfc, RngIntElt -> SeqEnum`

```
UseProj: BoolElt                    Default: true
```

For ordinary projective surface S, with only point singularities if S is not a hypersurface, returns a (possibly empty) sequence of homogeneous polynomials on the ambient of S that give the mth pluricanonical map on the blow-up desingularisation X of S for $m \in \mathbf{Z}$ [The length of the sequence is the dimension of $H^0(X, (\wedge^2\Omega_1(X))^{\otimes m})$]. If the ambient of S is $\mathbf{P}^n$, $n \geq 4$, this uses the base meromorphic 2-form and its divisor as computed by the previous intrinsic. If that hasn't already been computed and cached then it is computed first and the `UseProj` parameter has the same meaning as before. This intrinsic is essentially equivalent to `HomAdjoints(S,m,0)`.

Example: Canonical Forms and Plurigenera of a Del Pezzo `(ex-1afb7c)`

We give an example of the last two intrinsics applied to a singular degree 4 Del Pezzo that, again, will reappear in later examples.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> S := Surface(P4,[z^2+x*t-y*t-z*u, x*z-y*z+x*u]);
> dsds := ResolveSingularSurface(S);
> #dsds;
2
> B,F,w,div_dat := BasisOfHolomorphicTwoForms(S);
> B; F;
[]
u^2
```

Here there are no everywhere-defined 2-forms (as expected on a rational surface) as follows from the empty B, but we can also examine the base meromorphic 2-form and its divisor.

```
> w;
[
    $.1/$.2,
    $.5/$.2,
    1
]
```

The 2-form w is $dx_1 \wedge dx_4$ on the second affine patch.

```
> div_dat;
[*
    [
        <Scheme over Rational Field defined by
        x^2*t - 2*x*y*t + y^2*t - y*z*u + 2*x*u^2,
        x*z - y*z + x*u,
        z^2 + x*t - y*t - z*u,
        x - y,
        z,
        u, 4>,
        <Scheme over Rational Field defined by
        x*t + 2*u^2,
        y,
```

(continues on next page)

(continued from previous page)

```
        z + u, -3>,
        <Scheme over Rational Field defined by
        x,
        y,
        z - u, -1>,
        <Scheme over Rational Field defined by
        x,
        y,
        z, -1>
    ],
    <[*
        Object containing data about the exceptional divisors over singularity
        Scheme over Rational Field defined by
        $.1 - 1,
        $.2,
        $.3,
        $.4
        of algebraic surface
        Surface over Rational Field defined by
        $.1*$.3 + $.2^2 - $.2*$.4 - $.3,
        $.1*$.2 + $.1*$.4 - $.2,
        Object containing data about the exceptional divisors over singularity
        Scheme over Rational Field defined by
        $.1,
        $.2,
        $.3,
        $.4
        of algebraic surface
        Surface over Rational Field defined by
        $.1 - $.2 + $.3^2 - $.3*$.4,
        $.1*$.3 + $.1*$.4 - $.2*$.3
    *], [
        [ 2 ],
        [ -2, 1 ]
    ]>
*]
```

w vanishes to order 4 on (the strict transform of) a line [(-1)-curve] on S and has poles at a conic (to order 3) and two other line components (to order 1) of the $y = 0$ hyperplane divisor of S. It vanishes to order 2 on the single irreducible blow-up divisor over the singular point represented by $dsds[1]$ and has a pole of order 2 and zero of order 1 respectively at the two irreducible blow-up divisors over the other singular point, represented by $dsds[2]$.

We check that the (-1)-pluricanonical map is just the inclusion map of S into $\mathbf{P}^4$ up to a linear change of coordinates. This is because S is the anticanonical embedding of the desingularised weak Del Pezzo.

```
> B := PluriCanonicalBasis(S,-1);
> B;
[
    x*u - 2/3*y*u,
    y*z - 2*y*u,
    z^2 - 4*u^2,
    z*t - 2*t*u,
    z*u - 2*u^2
]
> // All elements of B are divisible by z-2u except the first.
```

(continues on next page)

(continued from previous page)

```
> // Change B[1] by an element of Ideal(S) to also give the divisibility.
> B[1] := B[1]-(1/3)*(x*z - y*z + x*u);
> B;
[
    -1/3*x*z + 1/3*y*z + 2/3*x*u - 2/3*y*u,
    y*z - 2*y*u,
    z^2 - 4*u^2,
    z*t - 2*t*u,
    z*u - 2*u^2
]
> [b div (z-2*u) : b in B];
[
    -1/3*x + 1/3*y,
    y,
    z + 2*u,
    t,
    u
]
```

FirstChernClassOfDesingularization(S): Srfc -> RngIntElt[RngMPolElt]

Returns the self-intersection of the canonical divisor, $K.K$ or K^2, on the blow-up desingularisation of surface S. Again, S must be an ordinary projective surface with only point singularities. If the desingularisation (by blow-up) of S hasn't already been cached, it is computed and cached at the start.

Note that this is not a birational invariant of desingularisations. It depends on the desingularisation that Magma computes. It gives a measure of how non-minimal the desingularisation is.

CanonicalIntersection(S, D): Srfc, DivSchElt -> RngIntElt

CanonicalIntersection(S, D): Srfc, Sch -> RngIntElt

CanonicalIntersection(dsd, i): DesingData, RngIntElt -> RngIntElt

```
MultsAndInts: Verbose                    Default: 1
```

The first two intrinsics return the intersection number $\tilde{D}.K$ of the strict transform $\tilde{D}$ of divisor D with the canonical divisor on the blow-up desingularisation of surface ordinary projective surface S. Recall that the strict transform of a divisor is the pull-back of the divisor to the blow-up with all blow-up divisor components removed (equivalently, the Zariski closure in the desingularisation of the divisor with the singular points of S removed). The second intrinsic is for an effective divisor which is given as a subscheme of S.

The third intrinsic return the intersection number with the canonical divisor of the i-th irreducible blow-up divisor over the surface singular point p that is represented by desingularisation data object *dsd*.

ExceptionalDivisors(S): Srfc -> List, List

```
UseSheaf: Verbose                    Default: true
Kodim   : Verbose                    Default: 10
```

For an ordinary projective surface, S, with only point singularities, and of Kodaira dimension ≥ 0 (equivalently, S is not rational or birationally ruled), returns two lists, the first containing all of the (-1)-curves on the blow-up desingularisation X of S (including irreducible divisors that split into a union of (-1)-curves over the algebraic closure of the base field), and the second containing all other irreducible curves on X that are contracted to points in the map from X to its minimal model.

Each of the two lists returned is itself a list containing two sequences. The first sequence consists of curves that are strict transforms of irreducible curves D on S. These are just returned

as the subschemes of S equal to D. The second sequence consists of irreducible blow-up divisors. These are given as sequence pairs $[i, j]$, representing the jth irreducible divisor of the ith desingularisation data structure `dsd` = `dsds`$[i]$ of S, where `dsds` is the desingularisation list returned by `ResolveSingularSurface`. (See example below and also the first example of the extended examples subsection).

The method is based upon finding all irreducible divisors common to a basis of global sections of a sufficiently large multiple of the canonical divisor. When S is singular, this is based upon computations using the base meromorphic divisor returned by `BasisOfHolomorphicTwoForms`. When S is already non-singular it can be quicker to use the canonical *sheaf*. To use this sheaf in the non-singular case, the parameter `UseSheaf` should be set to `true` (the default).

The smallest multiple of the canonical divisor required and some other issues require that the exact Kodaira dimension of S is known. If this is known to the user, and not necessarily already cached in the surface data, they can enter it by setting the `Kodim` parameter to a value between 0 and 2 inclusive. Any other value is ignored and the Kodaira dimension is then computed at the start.

Example: Exceptional Curves on a K3 Surface `(ex-838890)`

We compute the exceptional curves on a (non-minimal) K3 surface that is analysed in more detail later in the second extended example.

```
> P<a,b,c,d> := ProjectiveSpace(Rationals(),3);
> S := Surface(P, a^6+2*a^5*b-2*a^4*b^2+2*a^2*b^4+b^6+a^4*d^2+2*a^3*b*d^2+
>   a^2*b^2*d^2-a*b^3*d^2-2*b^4*d^2-a^2*b*d^3-2*a*b^2*d^3+2*a^2*d^4+b^2*d^4-
>   c^2*d^4-a*d^5);
> dsds := ResolveSingularSurface(S);
> #dsds; // only one singular point at [0,0,1,0]
1
> l1,l2 := ExceptionalDivisors(S);
> l1;
[*
    [
        Scheme over Rational Field defined by
        a^5 + a^4*b - 3*a^3*b^2 + 3*a^2*b^3 - a*b^4 + b^5,
        d,
        Scheme over Rational Field defined by
        a + b,
        d
    ],
    []
*]
> l2;
[*
    [],
    [
        [ 1, 1 ],
        [ 1, 2 ]
    ]
*]
```

So there are 6 (-1)-curves on the desingularisation X of S : 5 Galois-conjugate ones given by the first entry in $l_1[1]$, which is easily seen to split into 5 distinct lines over an extension of $\mathbf{Q}$, and one defined over $\mathbf{Q}$, given by the line $l_1[1][2]$. There are no blow-up (-1)-curves. On the other hand l_2 shows that, in minimising X, there are irreducible blow-up divisors that are also contracted. These consist of the first and second of the three irreducible blow-up divisors over the singularity represented by the first (and only!) desingularisation data object $dsds[1]$. Further analysis (see extended example 2 below) shows that the first irreducible blow-up divisor splits into 5 conjugate (-2)-curves and the second is a (-2)-curve defined over $\mathbf{Q}$.

126.4.5 Dual Resolution Graphs

For a blow-up desingularisation of a surface, functionality is now provided (as of Magma V2.25) to produce quite complete information on the configuration of exceptional divisors over a singular cluster-point, along with additional data about the divisors, in the form of a dual resolution graph. Using this data, we have also added intrinsics to compute the total number of points on the "blow-up fibre" over a singular point or the zeta-function of this fibre, for surfaces over finite fields. This is useful for point-counting on the desingularised surface. There are also versions of the intrinsics to compute the graphs and perform the point-counting for the *minimal* blow-up resolution, which is the initial blow-up resolution with any (-1)-curves amongst the exceptional divisors iteratively blown back down until no more remain.

`DualResolutionGraph(dsd): DesingData -> GrphMultUnd, SeqEnum, SeqEnum`

The dual resolution graph of a blow-up desingularisation over a singular point p of a surface, represented by the `dsd` desingularisation data object argument, is a multi-edge graph in which the nodes represent the irreducible divisors lying over p and edges represent intersections between these divisors. In addition, singular points of a divisor are represented by edge-loops from the corresponding node to itself. The graph, along with associated data about the divisors in node and edge labels (to be described below), is computed and returned by this intrinsic. It is of Magma's `MultiGraph` type. Once computed, it is stored for immediate return by later calls to the intrinsic.

There are two further return values that completely specify the intersection configuration. The first is a sequence of pairs of edges giving *singular intersections*: intersections between divisors where the intersection point is a singular point on (at least) one of the divisors. The corresponding entry for this would be a sequence pair $[e, f]$ where e is the graph loop-edge representing the singular point and f is the non-loop edge between the divisor nodes representing the intersection. If there was an intersection between divisors where the intersection point was a singular point on both divisors, there would be two entries in the sequence: $[e_1, f]$ and $[e_2, f]$. Similarly, multiple intersections involving one or more singular points would give several entries in the obvious way.

The other sequence contains multiple-intersections, i.e. intersection points that lie on 3 or more blow-up divisors. The entry for a multiple-intersection is a sequence containing each edge that represents the intersection between 2 divisors at that point. Thus, if $n \geq 3$ divisors meet at the point, the entry will be a sequence containing $n(n-1)/2$ graph edges.

Note that both of these sequences will be empty in the vast majority of cases.

Labels: The labels of the edges and nodes of the graph give information about the isomorphism types of the divisors and their canonical intersections and also relates the intersection points to the local blow-up affine patches accessible through `dsd`. This information is largely available through the accessor functions below, but it may be useful to the user to obtain it directly through the graph labels.

Node labels: the label for a node representing irreducible blow-up divisor D, is a list `[* s1,s2, s3,s4 *]` containing three elements.

- s_1 is the self-intersection number of D, $D.D$
- s_2 is the function field (type `FldFunFracSch`) of D [computed from a blow-up affine patch of D], which determines the curve normalisation of D
- s_3 is the canonical intersection number of D, $D.K$
- s_4 is a list of three elements $[*pi, ps, pls*]$. pi is the blow-up patch index of `dsd` which gives the representative affine patch of D from which s_2 is computed. The other two elements give the finite set of scheme-theoretic points of D off this patch. ps contains a list of representatives for these points on other blow-up patches in the form $< p, j >$ where p is the cluster giving the point on blow-up patch index j of `dsd`. pls is a sequence of places of function field s_2 corresponding to these points. [see examples below]

Edge labels 1: the label for a non-loop edge, representing an intersection p between blow-up divisors D_1 and D_2 is a list `[* s1,s2,s3,s4 *]` containing 4 elements.

- s_1 is the local intersection number $(D_1.D_2)_p$

- s_2 is a sequence of 3 indices $[i, j_1, j_2]$. The meaning is that p is a (non-singular) point on the blow-up surface patch index i of `dsd` and that j_1, j_2 are the indices in the sequence of blow-up divisors for patch i of D_1, D_2 [see examples below]
- s_3 is p as a cluster on the blow-up surface patch index i.
- s_4 is a 2-element list $[*l_1, l_2*]$. l_i is a sequence containing all of the places of the function field of D_i over the point p. If p is non-singular on D_i, l_i will only contain one place. This determines p "globally" on D_i.

Edge labels 2: the label for a loop edge, representing a singular point p on blow-up divisor D is a list `[* s1,s2,s3 *]` containing 3 elements.

- s_1 is a cluster p on one of the blow-up patches of `dsd` giving the singular point
- s_2 is an integer i, giving the index of the blow-up patch that p is on.
- s_3 is a sequence of all of the places of the function field of D over p.

`MinimalDualResolutionGraph(dsd): DesingData -> GrphMultUnd, SeqEnum, SeqEnum, Mtrx`

This is analogous to `DualResolutionGraph`, but gives the graph for the *minimal* resolution over singular point p of a surface represented by the `dsd` desingularisation data object argument. The minimal resolution is the initial (blow-up) resolution followed by the iterative contraction (blowing-down) of any (-1)-curves amongst the irreducible blow-up exceptional curves, until no more exist. The minimal resolution is "canonically" associated to the singularity and does not depend on the initial method of resolution (which is by iterated blow-up here).

Note that blowing down a (-1)-curve increases the self-intersection number of any curve intersecting it, so the minimisation process doesn't just involve (Galois-conjugate sets of) curves with self-intersection -1 in the initial resolution. For example, if a (-1)-curve C_1 intersects a (-2)-curve C_2 transversely in a single point, then, after blowing down C_1, C_2 becomes a new (-1)-curve.

The return values are the same as for `DualResolutionGraph` with an additional fourth return value, which is the intersection matrix of the irreducible exceptional divisors in the minimal resolution, taken in the order in which they appear as nodes in the minimal graph.

The node and edge labels in the minimal graph are the same as for the full dual resolution graph except with small changes for edges representing divisor singularities and intersections that are contractions of divisors in the minimisation process. For these, the cluster on a blow-up patch representing the singularity/intersection point is replaced by a string $cont_n$ where $n = 1, 2, \ldots$ is the stage of contractions at which the point is produced (in the minimisation process, a maximal disjoint set of (-1)-curves is considered to be contracted at each stage).

In practice, the original blow-up resolution in already minimal in the vast majority of cases.

Example: Dual Resolution Graph of a Degree 9 Surface `(ex-b5837e)`

We take a degree 9 hypersurface X with 2 non-simple singularities and some conjugate pairs of simple singularities that will be analysed in much more detail in one of the later extended examples. We consider the dual resolution graph of the less complex of the 2 non-simple singularities, which has 11 irreducible blow-up components in its resolution.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> S := Surface(P,
> z^2*t^7-x^8*t-8*x^7*t*y-4*x^6*t^2*y-24*x^6*t*y^2-24*x^5*t^2*y^2-
> 4*x^6*y^3-32*x^5*t*y^3-52*x^4*t^2*y^3-24*x^5*y^4-32*x^4*t*y^4-
> 48*x^3*t^2*y^4-52*x^4*y^5-64*x^3*t*y^5-16*x^2*t^2*y^5-48*x^3*y^6-
> 96*x^2*t*y^6-16*x^2*y^7-64*x*t*y^7-16*t*y^8);
> dsds,typ := ResolveSingularSurface(S);
> typ;  // resolution of blow-up type
2
> #dsds; // 5 Galois-conjugate sets of singular points
5
```

(continues on next page)

(continued from previous page)

```
> dsd := dsds[4];
> IntersectionMatrix(dsd);
[-2  0  0  0  0  0  0  1  0  0  0]
[ 0 -2  0  0  0  0  0  1  0  0  0]
[ 0  0 -2  0  0  0  0  0  1  0  0]
[ 0  0  0 -2  0  0  0  0  1  0  0]
[ 0  0  0  0 -2  0  0  0  0  0  1]
[ 0  0  0  0  0 -2  0  0  0  1  0]
[ 0  0  0  0  0  0 -2  0  0  1  0]
[ 1  1  0  0  0  0  0 -2  0  0  1]
[ 0  0  1  1  0  0  0  0 -2  0  1]
[ 0  0  0  0  0  1  1  0  0 -2  1]
[ 0  0  0  0  1  0  0  1  1  1 -4]
> G,sngs,multi_ints := DualResolutionGraph(dsd);
> G;
Multigraph
Vertex  Neighbours

1       8 ;
2       8 ;
3       9 ;
4       9 ;
5       11 ;
6       10 ;
7       10 ;
8       11 2 1 ;
9       11 4 3 ;
10      11 7 6 ;
11      10 9 8 5 ;

> sngs; multi_ints;
[]
[]
> vG := VertexSet(G);
> Label(vG.11);
[* -4,
    Function Field of Curve over Rational Field defined by
    $.2,
    $.3,
    2, [* 6, [*
        <Scheme over Rational Field defined by
        $.1 + 1/2,
        $.2,
        $.3, 10>,
        <Scheme over Rational Field defined by
        $.1 + 1,
        $.2,
        $.3, 13>,
        <Scheme over Rational Field defined by
        $.1,
        $.2,
        $.3, 16>
    *], [*
        [
            Place at (-1/2 : 0 : 0 : 1)
        ],
```

(continues on next page)

(continued from previous page)

```
        [
            Place at (-1 : 0 : 0 : 1)
        ],
        [
            Place at (1 : 0 : 0 : 0)
        ]
    *] *]
*]
> Edges(vG.11,vG.11);
[]
```

The above shows that the 11-th blow-up divisor D is a non-singular (no edges to itself) rational curve (from the function field) with self-intersection -4 and canonical intersection 2. It's representative affine patch is the 6-th one of the blow-up patches of the `dsd` resolution and there are three (rational) points not on that patch, with chosen representatives on the 10-th, 13-th, and 16-th blow-up patches of `dsd` and corresponding places of its function field as listed. We can get some of this data using intrinsics described below

```
> SelfIntersection(G,11);
-4
> CanonicalIntersection(G,11);
2
> ArithmeticGenus(G,11);
0
```

We can use the `dsd` accessor intrinsics to look at the actual representative affine patch, Da, of D

```
> dat11 := [e : e in BlowUpDivisorAllPatches(dsd,11) | e[4] eq 6][1];
> Da := dat11[1]; Da;
Curve over Rational Field defined by
1/4*$.2^2 - 1/4*$.3^2 + $.2,
$.2,
$.3
> Ambient(Da);
Affine Space of dimension 3 over Rational Field
Variables: $.1, $.2, $.3
```

Da is a line in affine 3-space and the function field of D has the usual projective closure of Da as 'its curve'. The first 2 of the list of 3 places for points of D off the representative patch appear to actually lie on Da. These can only be illegitimate representations of points on the blow-up divisor D because they are singular points on the 6th surface blow-up patch. This is easily checked:

```
> Xa6 := dat11[2]; // the 6th surface blow-up patch
> Xa6;
Scheme over Rational Field defined by
$.1^8*$.2^3 + $.1^8*$.2^2 + 3*$.1^7*$.2^3 + 4*$.1^7*$.2^2 + 13/4*$.1^6*$.2^3 +
6*$.1^6*$.2^2 + 3/2*$.1^5*$.2^3 + 4*$.1^5*$.2^2 + 1/4*$.1^4*$.2^3 +
2*$.1^4*$.2^2 + $.1^4*$.2 + 2*$.1^3*$.2^2 + 3*$.1^3*$.2 + 3/2*$.1^2*$.2^2 +
13/4*$.1^2*$.2 + 1/2*$.1*$.2^2 + 3/2*$.1*$.2 + 1/16*$.2^2 - 1/16*$.3^2 + 1/4*$.2
> IsSingular(Xa6![-1/2,0,0]); IsSingular(Xa6![-1,0,0]);
true (-1/2, 0, 0)
true (-1, 0, 0)
```

From the graph, D intersects the 5th, 8th, 9th, and 10th blow-up divisors. We look at its intersection(s) with the 10th.

```
> es := Edges(vG.10,vG.11); es; // single intersection point
[ < {10, 11}, 19 > ]
> e := es[1];
> Label(e);
[* 1,
    [ 16, 2, 1 ],
    Scheme over Rational Field defined by
    $.1,
    $.2,
    $.3,
    [*
        [
            Place at (0 : 1 : 0 : 0)
        ],
        [
            Place at (1 : 0 : 0 : 0)
        ]
    *]
*]
> // Local intersection multiplicity is 1 (its a transverse intersection).
> // Can also get this by
> LocalIntersectionNumber(e);
1
```

Example: Resolution Graph with Three Meeting Divisors (ex-58237d)

We give an example of a surface singularity whose resolution has 3 divisors all meeting together in a single point.

```
> A<a,b,c> := AffineSpace(Rationals(),3);
> S := Surface(A,-a^6-6*a^5*b-12*a^4*b^2-2*a^4*c-8*a^3*b^3-
>      8*a^3*b*c-8*a^2*b^2*c+a^2*b^2-a^2*c^2+2*a*b^3-2*a*b*c^2+
>      2*a*b*c+b^2*c+c^2);
> dsds := ResolveSingularSurface(S);
> #dsds;
1
> dsd := dsds[1];
> IntersectionMatrix(dsd);
[-2  1  1]
[ 1 -2  1]
[ 1  1 -4]
> G,sngs,multi_ints := DualResolutionGraph(dsd);
> G;
Multigraph
Vertex  Neighbours

1       3 2 ;
2       3 1 ;
3       2 1 ;
> sngs;
[]
> multi_ints;
[
    [ < {1, 2}, 1 >, < {1, 3}, 3 >, < {2, 3}, 5 > ]
]
```

The graph (or intersection matrix) shows that each blow-up divisor meets each of the other 2, but the only way (beyond detailed study of the graph labels) to distinguish between a triangular

configuration, where the three divisors don't have a common point, and the actual one, with a single intersection point between the 3, is via the `multi_ints` return value.

`SelfIntersection(dsd, i): DesingData, RngIntElt -> RngIntElt`

`SelfIntersection(G, i): GrphMultUnd, RngIntElt -> RngIntElt`

The self-intersection number $D.D$ of the ith irreducible blow-up divisor over the surface singularity whose blow-up resolution is represented by desingularisation data object `dsd`.

In the second version, the dual resolution graph `G` of the desingularisation is given as argument rather than `dsd`. In this version, `G` may also be the *minimal* dual resolution graph, in which case i refers to the ith divisor (node) of that graph.

`CanonicalIntersection(dsd, i): DesingData, RngIntElt -> RngIntElt`

`CanonicalIntersection(G, i): GrphMultUnd, RngIntElt -> RngIntElt`

The canonical intersection number, $D.K$, of the ith irreducible blow-up divisor over the surface singularity whose blow-up resolution is represented by desingularisation data object `dsd` with the canonical divisor K of the desingularised surface.

In the second version, the dual resolution graph `G` of the desingularisation is given as argument rather than `dsd`. In this version, `G` may also be the *minimal* dual resolution graph, in which case i refers to the ith divisor (node) of that graph.

`ArithmeticGenus(dsd, i): DesingData, RngIntElt -> RngIntElt`

`ArithmeticGenus(G, i): GrphMultUnd, RngIntElt -> RngIntElt`

The arithmetic genus of the ith irreducible blow-up divisor over the surface singularity whose blow-up resolution is represented by desingularisation data object `dsd`.

In the second version, the dual resolution graph `G` of the desingularisation is given as argument rather than `dsd`. In this version, `G` may also be the *minimal* dual resolution graph, in which case i refers to the ith divisor (node) of that graph.

`Genus(dsd, i): DesingData, RngIntElt -> RngIntElt`

`Genus(G, i): GrphMultUnd, RngIntElt -> RngIntElt`

The genus of the normalisation of the ith irreducible blow-up divisor over the surface singularity whose blow-up resolution is represented by desingularisation data object `dsd`.

In the second version, the dual resolution graph `G` of the desingularisation is given as argument rather than `dsd`. In this version, `G` may also be the *minimal* dual resolution graph, in which case i refers to the ith divisor (node) of that graph.

`LocalIntersectionNumber(e): GrphEdge -> RngIntElt`

The argument e should be the edge in a (minimal) dual resolution graph joining two different nodes, representing an intersection point p between irreducible exceptional divisors D and E. Return the local intersection number at p, $(D.E)_p$. (For two effective surface divisors D, E with no common component, $D.E$ is the sum over all intersection points of the local intersection numbers).

Point counting on resolution fibres

`NumberOfPointsOnResolutionFibre(dsd): DesingData -> RngIntElt`

For the blow-up resolution of a singular (scheme-theoretic) point p corresponding to desingularisation data object `dsd` on a surface over a finite field F, returns the number of F-rational points on the (reduced) fibre over p, which is the union of reduced exceptional divisors over p. Note that if p is not F-rational, this is 0. Uses the dual resolution graph of `dsd`.

`NumberOfPointsOnMinimalResolutionFibre(dsd): DesingData -> RngIntElt`

This is similar to the intrinsic `NumberOfPointsOnResolutionFibre` above, except it computes the number of points on the fibre of the *minimal* resolution over singular point p to which `dsd` corresponds. The minimal dual resolution graph is used.

ZetaFunctionOfResolutionFibre(dsd): DesingData -> FldFunRatUElt

With the same assumptions as for NumberOfPointsOnResolutionFibre, this intrinsic returns the zeta-function of the reduced resolution fibre over the singular point to which dsd corresponds. Again, the dual resolution graph of dsd is used.

ZetaFunctionOfResolutionFibre(dsd): DesingData -> FldFunRatUElt

With the same assumptions as for NumberOfPointsOnResolutionFibre, returns the zeta-function of the reduced resolution fibre over the singular point to which dsd corresponds. Again, uses the dual resolution graph.

ZetaFunctionOfMinimalResolutionFibre(dsd): DesingData -> FldFunRatUElt

As above, except computes the zeta-function of the reduced fibre of the *minimal* resolution over singular point p to which dsd corresponds. Uses the minimal dual resolution graph.

Example: Zeta Functions of Resolution Fibres (ex-ee12c0)

We count points and compute the zeta-functions of the fibres over the surface singularities of the last 2 examples over a finite field.

```
> P<x,y,z,t> := ProjectiveSpace(GF(37),3);
> S := Surface(P,
> z^2*t^7-x^8*t-8*x^7*t*y-4*x^6*t^2*y-24*x^6*t*y^2-24*x^5*t^2*y^2-
> 4*x^6*y^3-32*x^5*t*y^3-52*x^4*t^2*y^3-24*x^5*y^4-32*x^4*t*y^4-
> 48*x^3*t^2*y^4-52*x^4*y^5-64*x^3*t*y^5-16*x^2*t^2*y^5-48*x^3*y^6-
> 96*x^2*t*y^6-16*x^2*y^7-64*x*t*y^7-16*t*y^8);
> dsds := ResolveSingularSurface(S);
> // one of the pairs of conjugate simple singularities has split over GF(37)
> #dsds;
6
> [NumberOfBlowUpDivisors(dsd) : dsd in dsds];
[ 1, 1, 3, 3, 11, 38 ]
> dsd := dsds[5];
> ZetaFunctionOfResolutionFibre(dsd);
1/(177917621779460413*t^12 - 23081204987605675 2*t^11 + 60042323785325574*t^10 -
  7727454798626200*t^9 + 610886629350855*t^8 - 32512885054848*t^7 +
  1217402509092*t^6 - 32655381264*t^5 + 626830875*t^4 - 8433040*t^3 +
  75702*t^2 - 408*t + 1)
> Factorisation(Denominator($1));
[
    <$.1 - 1, 1>,
    <37*$.1 - 1, 11>
]
```

This is easily seen to be the zeta-function of 11 projective lines over $GF(37)$ with 10 intersection points.

```
> NumberOfPointsOnResolutionFibre(dsd);
408
> A<a,b,c> := AffineSpace(GF(37),3);
> S := Surface(A,-a^6-6*a^5*b-12*a^4*b^2-2*a^4*c-8*a^3*b^3-
>      8*a^3*b*c-8*a^2*b^2*c+a^2*b^2-a^2*c^2+2*a*b^3-2*a*b*c^2+
>      2*a*b*c+b^2*c+c^2);
> dsds := ResolveSingularSurface(S);
> #dsds;
> dsd := dsds[1];
> ZetaFunctionOfResolutionFibre(dsd);
1/(50653*t^4 - 54760*t^3 + 4218*t^2 - 112*t + 1)
> Factorisation(Denominator($1));
[
```

(continues on next page)

(continued from previous page)

```
    <$.1 - 1, 1>,
    <37*$.1 - 1, 3>
]
> NumberOfPointsOnResolutionFibre(dsd);
112
```

126.4.6 Extended Examples

Example: Desingularising a Degree 9 Elliptic Surface (ex-ef12b8)

The first example for blow-up desingularisation is a singular degree 9 hypersurface model S of an elliptic fibration X of general type (Kodaira dimension 1) which was sent in by a user. This has rather nasty singularities that are unsuitable for formal desingularisation, which rapidly grinds to a halt in trying to process them. However, the blow-up method can resolve all singularities in a few seconds. There are 2 conjugate A_3 singularities over $\mathbf{Q}(i)$, 2 pairs of conjugate A_1 singularities over $\mathbf{Q}(\sqrt{2})$, and 2 non-simple singularities over $\mathbf{Q}$. One of the non-simple singularities has 11 non-singular rational curves lying over it in the resolution - 10 (-2)-curves and a (-4)-curve - and the other has 38 (!) - 33 (-2)-curves, 4 (-3)-curves and a (-4)-curve. The geometric genus of X is 3 and the (rational) canonical map maps X onto a non-singular conic in $\mathbf{P}^2$. This gives the elliptic fibration map of X.

We show how we can quickly compute all this with the surface intrinsics, along with the intersection matrices of the curves (which give the intersection graphs) over each conjugate set of singular points.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> S := Surface(P,
> z^2*t^7-x^8*t-8*x^7*t*y-4*x^6*t^2*y-24*x^6*t*y^2-24*x^5*t^2*y^2-
> 4*x^6*y^3-32*x^5*t*y^3-52*x^4*t^2*y^3-24*x^5*y^4-32*x^4*t*y^4-
> 48*x^3*t^2*y^4-52*x^4*y^5-64*x^3*t*y^5-16*x^2*t^2*y^5-48*x^3*y^6-
> 96*x^2*t*y^6-16*x^2*y^7-64*x*t*y^7-16*t*y^8);
> dsds,typ := ResolveSingularSurface(S);
> typ;  // resolution of blow-up type
2
> #dsds; // 5 conjugacy classes of singular points
5
> IntersectionMatrix(dsds[1]); // 2 conjugate A3 singularities
[-4  0  2]
[ 0 -4  2]
[ 2  2 -4]
> IntersectionMatrix(dsds[2]); // 2 conjugate A1 singularities
[-4]
> IntersectionMatrix(dsds[3]); // 2 more conjugate A1 singularities
[-4]
> IntersectionMatrix(dsds[4]); // 11-component singular point
[-2  0  0  0  0  0  0  1  0  0  0]
[ 0 -2  0  0  0  0  0  1  0  0  0]
[ 0  0 -2  0  0  0  0  0  1  0  0]
[ 0  0  0 -2  0  0  0  0  1  0  0]
[ 0  0  0  0 -2  0  0  0  0  0  1]
[ 0  0  0  0  0 -2  0  0  0  1  0]
[ 0  0  0  0  0  0 -2  0  0  1  0]
[ 1  1  0  0  0  0  0 -2  0  0  1]
[ 0  0  1  1  0  0  0  0 -2  0  1]
[ 0  0  0  0  0  1  1  0  0 -2  1]
[ 0  0  0  0  1  0  0  1  1  1 -4]
```

Computing the self-intersections for the 38 components over the final singularity takes about five

minutes, so we don't include it here. The computation of the intersections between different components is still very fast.

```
>  // 38-component singular point
> IntersectionMatrix(dsds[5] : get_self_ints := false);
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1]
[1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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 0 0 0 0 0 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 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0]
> GeometricGenusOfDesingularization(S);
3
```

A sequence of degree 5 polynomials defining (rationally) the canonical map is returned by HomAdjoints(1,0,S). For simplicity, we will then just write down an equivalent map to $\mathbf{P}^1$, which is obvious from inspection, factoring out the common t^3 and combining with the standard birational projection of the image ($ac = b^2$ in $\mathbf{P}^2(a, b, c)$) to $\mathbf{P}^1$. We then check that this is an elliptic fibration map by looking at a particular fibre, which is an irreducible genus one curve.

```
> HomAdjoints(1,0,S);
[
    x^2*t^3,
    x*y*t^3,
    y^2*t^3
]
```

(continues on next page)

(continued from previous page)

```
> P1<u,v> := Curve(ProjectiveSpace(Rationals(),1));
> fib_map := map<S->P1|[x,y]>;
> bs := ReducedSubscheme(BaseScheme(fib_map));
> bs;
Scheme over Rational Field defined by
z*t,
x,
y
> // bs is just the two rational singular points
> f1 := Complement((P1![1,1])@@fib_map,bs);
> GroebnerBasis(f1);
[
    y^9 + 337/144*y^8*t + y^7*t^2 - 1/144*z^2*t^7,
    x - y
]
> IsIrreducible(f1); IsReduced(f1);
true
true
> Genus(Curve(f1));
1
```

We can go further here and find out exactly the structure of bad fibres of the minimal model of X and how they relate to the graphs of exceptional divisors over the singular points. Firstly, we check that our desingularisation is not minimal by computing its first Chern class. This is the self-intersection number K^2 where K is a canonical divisor on the desingularisation. It turns out to be -27. As K^2 for the minimal model is 0, the desingularisation is the minimal model blown up 27 times. Normally, the intrinsic `FirstChernClassOfDesingularization` would be called here. As that uses full intersection matrices for the exceptional blow-up divisors and we have avoided computing the fifth one fully to save time, we cheat by manually inserting the diagonal entries of the fifth matrix and then perform the computation undertaken by the intrinsic directly.

```
> imats := [*IntersectionMatrix(dsds[i]) : i in [1..4]*];
> imat5 := IntersectionMatrix(dsds[5] : get_self_ints := false);
> imat5 := imat5 + DiagonalMatrix([-2 : i in [1..33]] cat [-3,-3,-3,-4,-3]);
> Append(~imats,imat5);
> dms := [DifferentialMultiplicities(dsd) : dsd in dsds];
> i2 := &+[(m*imats[i]*Transpose(m))[1,1] where m is Matrix(Vector(dms[i])) :
>              i in [1..5]];
> d := Degree(S); // d=9
> K2 := d*(d-4)^2+i2;
> K2;
-27
```

We can find the irreducible divisors that are contracted to give the minimal model by calling the `ExceptionalDivisors` intrinsic. This is a little slow in this complex example but still only takes about a minute and a half on our machine.

```
> time m1s,others := ExceptionalDivisors(S);
Time: 92.490
> m1s; // The (-1)-curves
[*
    [
        Scheme over Rational Field defined by
        x + 2*y,
        t,
        Scheme over Rational Field defined by
```

(continues on next page)

(continued from previous page)

```
        x + y,
        t,
        Scheme over Rational Field defined by
        x,
        t,
        Scheme over Rational Field defined by
        y,
        t
    ],
    []
*]
> others; // the other curves to be contracted
[*
    [],
    [
        [ 5, 1 ],
        [ 5, 2 ],
        [ 5, 3 ],
        [ 5, 4 ],
        [ 5, 5 ],
        [ 5, 6 ],
        [ 5, 7 ],
        [ 5, 8 ],
        [ 5, 9 ],
        [ 5, 10 ],
        [ 5, 19 ],
        [ 5, 20 ],
        [ 5, 21 ],
        [ 5, 22 ],
        [ 5, 25 ],
        [ 5, 26 ],
        [ 5, 27 ],
        [ 5, 28 ],
        [ 5, 29 ],
        [ 5, 30 ],
        [ 5, 31 ],
        [ 5, 32 ],
        [ 5, 33 ]
    ]
*]
```

Thus, the (strict transforms of the) four lines in `m1s[1]` are the (-1)-curves and those along with the 23 blow-up divisors over the fifth singular point whose indices are given by`others[2]` are the components that need to be contracted to give the minimal model for X. This tallies with the earlier K^2 computation (and renders it redundant!).

Further analysis shows the following.

The irreducible divisors above the 2 $\mathbf{Q}$-rational singular points map to one of the four points $0, -1, -2, \infty$ on $\mathbf{P}^1$ under *fib_map* except for the curve of self-intersection -3 over singularity 4 and the curve of self-intersection -4 over singularity 5, both of which map onto $\mathbf{P}^1$. This is computed using `BlowUpDivisors` to get affine patches of the divisors along with the maps down to S.

There are four chains of curves, each starting from one of these (-1)-curves, continuing down through (-2)-curves above the fifth singularity p and stopping at the four (-3)-curves over p. The blow-down from the desingularisation to the minimal model contracts these four chains, leaving only (-2)-curves and the (-4)-curve over p. We also find another 8 irreducible curves in S over the

four points of $\mathbf{P}^1$ mentioned above and these 8 curves are also (-2)-curves and join the remaining (non-blown-down) components over p to those over the fourth singularity. All of these (-2)-curves together - the 8 new ones and those over p and the fourth singularity, excluding the two (-4)-curves which map onto $\mathbf{P}^1$ - form four disjoint simple loops containing eight curves each. These are the fibres of the minimal model over the four points of $\mathbf{P}^1$ and are all of type I_8 in the usual Neron/Kodaira notation. The curves over the six simple singular points along with an additional (-2)-curve (strict transform of a curve in S) for each singularity form the remaining bad fibres on the minimal model which give two fibres of type I_4 and four of type I_2.

We can deduce from all the information gained that the divisor $H + K$ on the desingularisation (H is the hyperplane divisor and K the canonical) is base-point free and its associated map into projective space contracts precisely the curves on which all differentials vanish, so factors through the minimal model. It should give an isomorphic embedding of the minimal model into ordinary projective space. However, it maps into projective space of dimension 30, which is rather large.

We briefly illustrate a couple of these final points.

```
> C,_,mp := BlowUpDivisor(S,dsds[4],11);
> mp := Restriction(mp,C,S : Check := false);
> fmpC := Expand(mp*fib_map);
> // manually remove common factors from defining equations to
> // get the fibre map from C to P^1
> fmpC := map<C->P1|[e div g : e in DefiningPolynomials(fmpC)]> where
>   g is GCD(DefiningPolynomials(fmpC));
> C; fmpC;
Curve over Rational Field defined by
1/4*$.2^2 - 1/4*$.3^2 + $.2,
$.2,
$.3
Mapping from: Crv: C to Crv: P1
with equations :
1
$.1
> Image(fmpC); // It's onto!
Curve over Rational Field defined by
0
> C,_,mp := BlowUpDivisor(S,dsds[5],34);
> mp := Restriction(mp,C,S : Check := false);
> fmpC := Expand(mp*fib_map);
> fmpC := map<C->P1|[e div g : e in DefiningPolynomials(fmpC)]> where
>   g is GCD(DefiningPolynomials(fmpC));
> C; fmpC;
Curve over Rational Field defined by
$.2,
$.3
Mapping from: Crv: C to Crv: P1
with equations :
1
$.1*$.2 + 4*$.2 - 1/2
> Image(fmpC); // C maps to [-2:1]
Scheme over Rational Field defined by
1/2*u + v
> // |H+K| divisor map is the (1,1)-adjoint map : HomAdjoints gives
> // defining polynomials
> HKeqns := HomAdjoints(1,1,S);
> #HKeqns; // It's into P^30
31
```

Example: K3 Double Covers with Tritangent Lines (ex-5e9e06)

This example deals with $K3$ surfaces S that are double covers of the plane ramified over smooth sextics. If the plane sextic f_6 has a tritangent line, then the pull back of that line to S splits into two rational (-2)-curves Z_1 and Z_2. General theory (see [SD74], Section 7) says that if B is the divisor class of the pullback of lines in the plane (the general member of which is a smooth genus 2 curve: a double cover of the plane line ramified over the six points where the line meets the ramification sextic) then the divisor map for the divisor class $2B + Z_1$ gives a birational embedding of S into $\mathbf{P}^6$ which is an isomorphism to its image X outside of contracting Z_1 to an A_1 singularity. The image X is the intersection of a singular cone over a Veronese surface in $\mathbf{P}^5$ with three cubics. Furthermore, the Veronese cone is generated by the quadrics in the ideal of X. The image of Z_1 is the vertex of the cone.

We will use the desingularisation machinery to analyse a singular model of such an S in $\mathbf{P}^3$, construct the divisor map on this model and check that the image has the predicted form. Note, generally that if the smooth sextic has equation $f_6(a, b, d) = 0$ in $\mathbf{P}^2(a, b, d)$ then a birational model for the double cover in $\mathbf{P}^3(a, b, c, d)$ is given by the degree 6 homogeneous equation $F = f_6(a, b, d) - c^2d^4 = 0$ and this hypersurface is only singular at the point with homogeneous coordinates $[0 : 0 : 1 : 0]$ so blow-up desingularisation may be applied. In the more general case of a sextic with simple curve singularities (which also leads to a $K3$ surface), there will be further singular points on the hypersurface model, but still only singular points (not curves) as long as coordinates are chosen so that none of the singularities of f_6 are at infinity (i.e. on the line $d = 0$).

Our example has

$f_6 = a^6 + 2a^5b - 2a^4b^2 + 2a^2b^4 + b^6 + a^4d^2 + 2a^3bd^2 + a^2b^2d^2 - ab^3d^2 - 2b^4d^2 - a^2bd^3 - 2ab^2d^3 + 2a^2d^4 + b^2d^4 - ad^5$

which is easily checked to be nonsingular and to have $a = 0$ as a tritangent line. We will take the pullback of the line $b = 0$ as an irreducible divisor of class B.

```
> P<a,b,c,d> := ProjectiveSpace(Rationals(),3);
> S := Surface(P, a^6+2*a^5*b-2*a^4*b^2+2*a^2*b^4+b^6+a^4*d^2+2*a^3*b*d^2+
>   a^2*b^2*d^2-a*b^3*d^2-2*b^4*d^2-a^2*b*d^3-2*a*b^2*d^3+2*a^2*d^4+b^2*d^4-
>   c^2*d^4-a*d^5);
> dsds := ResolveSingularSurface(S);
> #dsds; // only one singular point at [0,0,1,0]
1
> dsd := dsds[1];
> IntersectionMatrix(dsd);
[-10   0   5]
[  0  -2   1]
[  5   1  -4]
```

The first irreducible blow-up divisor D actually splits into 5 disjoint (-2)-curves over a finite extension of $\mathbf{Q}$. We can see this by examining the equations for it on a chosen affine patch. To check that the components don't intersect on another patch, we could use `BlowUpDivisorAllPatches`, but it suffices to show that the intersection number $K.D = 0$ (implying that $K.D_i = 0$ for each component : they are all conjugate and so have the same intersection number with the $\mathbf{Q}$-rational class K), and that each component is a rational curve. Then, if any component were singular on another patch or the components intersected, $D.D$ would be greater than -10. The other two classes are a further genus 0 nonsingular curve (also disjoint from D by the intersection matrix) and a non-singular genus 2 curve.

```
> C1 := BlowUpDivisor(S,dsd,1); C1;
Curve over Rational Field defined by
$.2^5 - $.2^4 + 3*$.2^3 - 3*$.2^2 + $.2 + 1,
$.1
> CanonicalIntersection(dsd,1);
0
> C2 := BlowUpDivisor(S,dsd,2); C2;
Curve over Rational Field defined by
```

(continues on next page)

(continued from previous page)

```
$.1,
$.2 + 1
> C3 := BlowUpDivisor(S,dsd,3); C3;
$.1^6*$.3^2 + 2*$.1^4*$.3^2 - 2*$.1^2*$.3^2 + 2*$.1*$.3^2 + $.3^2 - 1,
$.2
> Genus(Curve(C3));
2
```

As in the last example, we can get the geometric genus of the blow-up, a basis for holomorphic differentials, and also the set of irreducible divisors on which all these differentials vanish. Comparing with the K^2 intersection number verifies that the minimal $K^2 = 0$ and that the desingularisation is a minimal model blown up twelve times. We also confirm that S is (birationally) a $K3$ surface by checking that the second plurigenus and the arithmetic genus are 1. These invariant values, along with $p_g = 1$, only occur for blow-ups of $K3$ surfaces.

```
> GeometricGenusOfDesingularization(S);
1
> PlurigenusOfDesingularization(S,2);
1
> ArithmeticGenusOfDesingularization(S);
1
> diffs := HomAdjoints(1,0,S); diffs;
[
    d^2
]
> // Only one holomorphic differential up to a scalar.
> // Curves on which it vanishes are those in S on which
> // d vanishes and blowup divisors on which $d^2$ vanishes
> // to a higher order than the differential multiplicity.
> // This turns out to be the first 2 blow-up divisors.
> Dd := S meet Scheme(P,d);
> divs1 := IrreducibleComponents(ReducedSubscheme(Dd));
> divs1;
[
    Scheme over Rational Field defined by
    a^5 + a^4*b - 3*a^3*b^2 + 3*a^2*b^3 - a*b^4 + b^5,
    d,
    Scheme over Rational Field defined by
    a + b,
    d
]
> // The first divisor above splits into 5 disjoint lines
> // again over an extension of Q. We check that all 6
> // geometrically irreducible lines in divs1 have strict
> // transforms on the blow-up that are (-1)-curves by
> // computing that K.D=-1 for all of them
> CanonicalIntersection(S,divs1[1]);
-5
> CanonicalIntersection(S,divs1[2]);
-1
> DifferentialMultiplicities(dsd);
[ -3, -3, -6 ]
> [2*m : m in Multiplicities(S,dsd,Dd)];
[ 4, 4, 6 ]
> FirstChernClassOfDesingularization(S);
-12
```

The strict transform of each of the five conjugate lines in `divs1` meets one of the five conjugate lines of the first irreducible blow-up divisor once transversally. The line over $\mathbf{Q}$ in `divs1` meets blow-up divisor 2 once transversally. This is checked by computing intersection numbers. This shows that the blow-up process to go from the minimal K3 surface to our desingularisation is to blow up 6 points (one over $\mathbf{Q}$ and 5 conjugate ones over $\mathbf{Q}$) and then to further blow up a point on each of the (-1)-curves arising.

```
> _,ints := MultiplicitiesAndIntersections(S,dsd,divs1[1]);
> ints;
[ 5, 0, 0 ]
> _,ints := MultiplicitiesAndIntersections(S,dsd,divs1[2]);
> ints;
[ 0, 1, 0]
```

We now construct the $2B + Z1$ divisor map. The method is to first get a basis for the Riemann–Roch space of the divisor using the normal call to the divisor/sheaf intrinsic. However, as explained in the description of the `LinearSystemDivisorRestriction` intrinsic, we then have to impose extra conditions at the blow-up divisors. If the denominator of the R-R space vanishes to given multiplicities at these divisors, we have to take the subspace of numerators that also vanish to at least these multiplicities. As it is slightly more convenient to get the R-R space as a separate basis of numerators and denominators using the sheaf intrinsic, we use that here.

```
> // First get B and Z1
> // Z1 is one of the components of the pullback of a=0 to S
> cmps := IrreducibleComponents(S meet Scheme(P,a));
> cmps;
[
    Scheme over Rational Field defined by
    b^3 - b*d^2 + c*d^2,
    a,
    Scheme over Rational Field defined by
    b^3 - b*d^2 - c*d^2,
    a
]
> Z1 := cmps[1];
> // B is the pullback of b=0 to S
> B := S meet Scheme(P,b);
> GroebnerBasis(B);
[
    a^6 + a^4*d^2 + 2*a^2*d^4 - c^2*d^4 - a*d^5,
    b
]
> rr,den := RiemannRochBasis(S,
>  EquidimensionalPart((Ideal(B)^2*Ideal(Z1))+Ideal(S)));
> rr;
[
    a^3,
    a^2*b,
    a^2*c,
    a^2*d,
    a*b^2,
    a*b*c,
    a*b*d,
    a*c^2,
    a*c*d,
    a*d^2,
    b^3 - b*d^2 - c*d^2
]
```

(continues on next page)

(continued from previous page)

```
> den;
a*b^2
> // Do the linear system restriction at blow-up divisors
> mults := Multiplicities(S,dsd,S meet Scheme(P,den)); mults;
[3,3,6]
> div_map_eqns := LinearSystemDivisorRestriction(S,rr,[mults]:
>      CheckB := false);
> div_map_eqns;
[
    a^3,
    a^2*b,
    a^2*d,
    a*b^2,
    a*b*d,
    a*d^2,
    b^3 - b*d^2 - c*d^2
]
```

We finally check that the image is of the expected type.

```
> P6<x,y,z,s,t,u,v> := ProjectiveSpace(Rationals(),6);
> div_map := map<S->P6|div_map_eqns>;
> X := Image(div_map);
> Xeqns := MinimalBasis(Ideal(X));
> Xeqns;
[
    t^2 - s*u,
    z*t - y*u,
    z*s - y*t,
    z^2 - x*u,
    y*z - x*t,
    y^2 - x*s,
    x^2*z + 2*x^2*t - 2*x*y*t + 2*y*s*t + x*z*u + 2*x*t*u +
    y*t*u - s*t*u - y*u^2 + 2*z*u^2 - 2*s*u^2 - u^3 +
        2*s*t*v - 2*t*u*v - z*v^2,
    x^2*y + 2*x^2*s - 2*x*y*s + 2*y*s^2 + x*y*u + 2*x*s*u +
     y*s*u - s^2*u - y*t*u - 2*s*t*u + 2*y*u^2 - t*u^2 +
        2*s^2*v - 2*s*u*v - y*v^2,
    x^3 - 6*x^2*s + 4*x*y*s + 2*x*s^2 - 4*y*s^2 + x^2*u -
     3*x*s*u - 3*y*s*u + 2*s^2*u - x*t*u + 4*s*t*u +
       2*x*u^2 - 4*y*u^2 - z*u^2 + 2*t*u^2 + 2*y*s*v -
        4*s^2*v - 2*y*u*v + 4*s*u*v - x*v^2 + 2*y*v^2
]
```

As expected, there are 3 cubics. Also, the quadrics do not contain v so define a singular cone over the scheme they define in the $\mathbf{P}^5$ hyperplane $v = 0$. Can see by inspection that this is a Veronese surface. However, we can check this mechanically by the characterisation that it is a non-singular, irreducible, non-degenerate (not lying in a hyperplane) surface of degree 4 with $K^2 = 9$. We verify the K^2 condition with the general machinery for surfaces with at most simple singularities. We also check directly that X has only an A_1 singularity at the cone vertex, $[0 : 0 : 0 : 0 : 0 : 0 : 1]$.

```
> P5 := ProjectiveSpace(Rationals(),5);
> hm := hom<CoordinateRing(P6)->CoordinateRing(P5)|
>   [P5.i : i in [1..6]] cat [0]>;
> Xquad := Surface(P5,[hm(Xeqns[i]) : i in [1..6]]);
> // The surface constructor has checked irreducibility
```

(continues on next page)

(continued from previous page)

```
> // and dimension!
> IsNonsingular(Xquad);
true
> Degree(Xquad);
4
> ChernNumber(Xquad,1); // K^2 = 9
9
> // Check non-degeneracy : no linear form vanishes on Xquad
> Saturate(~Xquad);
> Min([TotalDegree(f) : f in Basis(Ideal(Xquad))]);
2 1
> HasOnlySimpleSingularities(X : ReturnList := true);
true [* <(0 : 0 : 0 : 0 : 0 : 0 : 1), "A", 1> *]
```

Example: Blowing Down Lines on a Degenerate Del Pezzo (ex-f69efd)

This example deals with blowing down lines that pass through singular points on a degenerate Del Pezzo surface. On non-singular Del Pezzos, a disjoint collection of lines $\{L_i\}$ can be blown down by applying the divisor map of the divisor class $H + \sum_i L_i$, where H is a hyperplane divisor. The divisor map can be obtained from either the sheaf `DivisorMap` intrinsic or from the Riemann–Roch basis computed using the divisor machinery.

In the singular case, the Riemann–Roch space computation in general produces a space of functions that is too large and which doesn't give the anti-canonical embedding of the blow-down. As in the last example, we can get the correct Riemann–Roch space for the blow-down map using linear system divisor restriction.

Our example is a degree 4 Del Pezzo S in $\mathbf{P}^4$ with an A_1 and an A_2 singularity. We have two lines L_1 and L_2 that we want to blow down which intersect at the A_2 singularity but whose strict transforms are (-1)-curves that do not intersect on the desingularised Del Pezzo. The Riemann–Roch space for $H + L_1 + L_2$ is of dimension one too large, but gives the correct anticanonical map into $\mathbf{P}^6$ after restriction.

Since we want the strict transform of our divisor on S to be a full divisor of the Cartier hyperplane divisor class added to the strict transforms of the two lines, we have to be careful to take a hyperplane on S that doesn't pass through the singular points. Otherwise, divisor restriction leads to a Riemann–Roch space that is too small: the subspace of the required space of functions that vanish to some positive multiplicity on the (-2)-curves above the singular points.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> S := Surface(P4,[z^2+x*t-y*t-z*u, x*z-y*z+x*u]);
> HasOnlySimpleSingularities(S : ReturnList := true);
true
[* <(0 : 0 : 0 : 1 : 0), "A", 2>, <(1 : 1 : 0 : 0 : 0), "A", 1> *]
> dsds := ResolveSingularSurface(S);
> L1 := Scheme(P4,[x,y,z]);
> L2 := Scheme(P4,[x-y,z,u]);
> MultiplicitiesAndIntersections(S,L1);
[
    [ 0 ],
    [ 1, 1 ]
]
[
    [ 0 ],
    [ 1, 0 ]
]
> MultiplicitiesAndIntersections(S,L2);
[
```

(continues on next page)

(continued from previous page)

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

We see from this that for the two (-2)-curves over the A_2 singularity, the strict transform of L_1 passes through one (transversally) and that of L_2 passes through the other. The strict transform of L_2 passes through the A_1 singularity and meets the (-2)-curve above it, whereas that of L_1 doesn't. In particular, the strict transforms of the two lines don't meet.

We first show that if we take a very bad hyperplane divisor ($x = 0$) that passes through both of the singularities, divisor restriction gives a Riemann–Roch space that is much too small.

```
> D := Divisor(S,L1)+Divisor(S,L2)+Divisor(S,Scheme(S,x));
> rr_fns := RiemannRochBasis(D);
> #rr_fns; // too large by one
8
> den := LCM([Denominator(f): f in rr_fns]);
> den;
y*z - x*u
> B := [R!(den*f) : f in rr_fns] where R is CoordinateRing(P4);
> ms := Multiplicities(S,Scheme(S,den)); ms;
[
    [ 1 ],
    [ 3, 2 ]
]
> B1 := LinearSystemDivisorRestriction(S,B,ms);
> #B1; // too small by 5!
2
```

We now get the correct result for blow-down to a non-degenerate degree 6 Del Pezzo using a hyperplane divisor that avoids the singular points ($x + t = 0$).

```
> D := Divisor(S,L1)+Divisor(S,L2)+Divisor(S,Scheme(S,x+t));
> rr_fns := RiemannRochBasis(D);
> #rr_fns; // too large by one again
8
> den := LCM([Denominator(f): f in rr_fns]);
> den;
y*z + z*t - x*u
> B := [R!(den*f) : f in rr_fns] where R is CoordinateRing(P4);
> ms := Multiplicities(S,Scheme(S,den)); ms;
[
    [ 1 ],
    [ 1, 1 ]
]
> B1 := LinearSystemDivisorRestriction(S,B,ms);
> #B1; // correct!
7
```

$B1$ gives the blow-down map to the anti-canonical image in $\mathbf{P}^6$.

```
> P6<a,b,c,d,e,f,g> := ProjectiveSpace(Rationals(),6);
> bdmp := map<S->P6|B1>;
```

(continues on next page)

(continued from previous page)

```
> X := Image(bdmp);
> MinimalBasis(Ideal(X));
[
    e*f - d*g,
    d*f - f^2 - a*g,
    c*f - b*g,
    d*e - c*g - f*g,
    a*e + a*g - b*g,
    d^2 - f^2 - b*g,
    c*d - b*e,
    a*d + a*f - b*f,
    a*c - b*d + b*f
]
> Degree(X);
6
> ChernNumber(X,1); //K^2 for X
6
```

126.5 Surfaces in $\mathbb{P}^3$

126.5.1 Introduction

This section describes several packages of functionality developed for working with (hyper)surfaces in three-dimensional projective space $\mathbf{P}^3$ which can have arbitrary singularities (at least in the characteristic 0 case). It was designed and implemented by Tobias Beck and has at its core, routines to compute a **formal desingularization** of 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. The algorithm is based on the method of Jung. It is fully described in [Bec07] and will be partly sketched out in the following two subsections.

As of Magma V2.21, the method of resolution of singularities by local blow-up is also available in the case that X has only point singularities. This can often be faster than the formal desingularisation in practice, although the method is less elegant, and provides additional information (see section *Desingularisation by Blow Up*). It is now the default method (given the point singularity restriction) when the general *ResolveSingularSurface* intrinsic is used. If desired, formal desingularisation can be used instead by calling that intrinsic with the `UseFormalDesing` parameter set to `true`, or the intrinsic `FormallyResolveProjectiveHypersurface` described below can be called. It should also be noted that formal resolution requires that the base field of the surface is characteristic 0. This is not required for blow-up resolution.

Major functionality to be described in this section, which is enabled by our resolutions, includes the computation of important birational invariants of *any* desingularization of X like the arithmetic and geometric genera and higher geometric plurigenera. Furthermore, we also provide an intrinsic for the computation of *(i,j)-adjoint* maps as rational maps on X. For formal desingularisation, the theoretical and algorithmic details may be found in [BS08]. Blow-up resolutions allow similar computations and the intrinsics for all of the above now work with either type of resolution data.

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 parametrization 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 [Sch00a]. As with the rest of the package, it was originally coded to make use of formal desingularisation data but now also works with blow-up resolution data.

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.

126.5.2 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 $\{p_1, \ldots, p_r\} \in Q$ be the generic points of the decomposition of D into irreducible components and $\{q_1, \ldots, q_s\} \in Q$ 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): RngMPolElt -> List, List, List, RngIntElt`

```
Factors : SeqEnum                    Default: []
Ps      : RngMPolElt                 Default: 0
Focus   : RngMPolElt                 Default: 0
ExtName : MonStgElt                  Default: "alpha"
ExtCount: RngIntElt                  Default: 0
Resolve : Verbose                    Default: 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: Embedded Resolution of an Affine Curve (ex-32b426)

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): RngMPolElt -> List, List, List, RngIntElt`

```
Focus   : RngMPolElt                    Default: 0
ExtName : MonStgElt                     Default: "alpha"
ExtCount: RngIntElt                     Default: 0
Resolve : Verbose                       Default: 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: Embedded Resolution of a Projective Curve (ex-f419e5)

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.

126.5.3 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 a parameter option with the main function *FormallyResolveProjectiveHyperSurface*, which removes "non-singular" morphisms and possibly others that have no effect on the computation of birational invariants and m-adjoint maps.

ResolveAffineMonicSurface(s): RngUPolElt -> List, RngIntElt

```
Focus   : RngMPolElt                    Default: 0
ExtName : MonStgElt                     Default: "alpha"
ExtCount: RngIntElt                     Default: 0
Resolve : Verbose                       Default: 1
```

The main user resolution function *FormallyResolveProjectiveHyperSurface* 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 *Adjoint Systems and Birational Invariants*).

One can specify a focus ideal $\mathbb{F} \subset \mathbf{E}[x,y]$ 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 $\mathbb{F} + \langle \mathrm{disc}_z(s) \rangle$.

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: Formal Desingularization of an Affine Surface (ex-455987)

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.

`FormallyResolveProjectiveHyperSurface(S): Srfc -> List, RngIntElt`

```
AdjComp : BoolElt                    Default: false
ExtName : MonStgElt                  Default: "gamma"
ExtCount: RngIntElt                  Default: 0
Resolve : Verbose                    Default: 1
```

The main intrinsic is now *ResolveSingularSurface* for resolution of singularities on surfaces and this can be forced to produce a formal resolution (if possible) by calling it with the parameter `UseFormalDesing` set to `true`. That is essentially equivalent to calling this intrinsic directly.

This gives a direct option to get the formal resolution and allows the use of the `ExtName` and `ExtCount` parameters.

It is similar in description to *ResolveAffineMonicSurface* above. The argument is a projective surface $S \subset \mathbf{P}^3_{\mathbf{E}}$ over a field of characteristic 0. Computes and returns 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 *Adjoint Systems and Birational Invariants*). 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].

The returned resolution list is also cached on S in a surface attribute (alongside any blow-up desingularisation data that has been computed). Thus, it will not be recomputed on a further call to this intrinsic or on calls to any other intrinsics using formal desingularisation data. The only time it may be recomputed is when this (or *ResolveSingularSurface*) is first called with `AdjComp true` and then called again with `AdjComp false`. S can cache *both* an 'adjcomp only' formal desingularisation list and a general one and the cached data is labelled correspondingly.

Example: Formal Desingularization of a Projective Surface `(ex-527dc0)`

Computing a formal desingularization is easy. We also see that setting `AdjComp` to `true` produces fewer formal divisors needed to only compute birational invariants or adjoint spaces.

```
> P<x,y,z,w> := ProjectiveSpace(Rationals(), 3);
> F := w^3*y^2*z+(x*z+w^2)^3;
> S := Surface(P, F);
> #FormallyResolveProjectiveHypersurface(S : AdjComp := true);
18
> desing := FormallyResolveProjectiveHypersurface(S); #desing;
26
> // could have used ResolveSingularSurface(S : UseFormalDesing := true);
```

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
```

(continues on next page)

(continued from previous page)

```
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 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 *Adjoint Systems and Birational Invariants*) 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

$$\begin{aligned}
&\frac{\mathtt{X}^5}{(\partial F/\partial w)(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})}\mathrm{d}\mathtt{Y}/\mathtt{X} \wedge \mathrm{d}\mathtt{Z}/\mathtt{X} \\
&= \frac{1}{(\partial F/\partial w)(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})}\mathrm{d}(-st^2) \wedge \mathrm{d}(-t^2) \\
&= \frac{1}{(\partial F/\partial w)(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})}(2st\mathrm{d}t + t^2\mathrm{d}s) \wedge 2t\mathrm{d}t \\
&= \frac{1}{(\partial F/\partial w)(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})}2t^3\mathrm{d}s \wedge \mathrm{d}t
\end{aligned}$$

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
```

126.5.4 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

$$\left(\frac{\partial F/\partial x_j}{x_i^{d-1}}\right)^{-1} \bigwedge_{k\in\{0,\ldots,3\}\setminus\{i,j\}} \mathrm{d}\frac{x_k}{x_i}$$

(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 $\mathbb{F}_{S,m} \subset (\Omega^{\wedge 2}_{\mathbf{E}(S)|\mathbf{E}})^{\otimes m}$ 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, when $m \geq 0$, and one can show $\mathbb{F}_{S,m} \subseteq (\omega_S^0)^{\otimes m}$. For more details we refer to [BS08].

Now since $\mathbb{F}_{S,m}$ 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 $\mathbb{F}_{S,m}$ 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 subsystem of the standard linear system of degree $n + m(d-4)$ homogeneous polynomials on S, corresponds to global sections of the Serre twist $\mathbb{F}_{S,m}(n)$. 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. The untwisted m-adjoint map is just the *m-pluri-canonical* map when $m > 0$ and the *(-m)-pluri-anticanonical* map when $m < 0$.

Note that the computations are all valid for $m < 0$ but in that case, the adjoint and twisted adjoint linear systems *do* depend on the desingularisation, so the results are not birationally invariant but are specific to the particular formal or blow-up desingularisation that has been computed.

The intrinsics below have all been updated so that they can either use formal desingularisation data or the newer blow-up desingularisation data. Once a singularity resolution has been computed, the desingularisation data is cached with the hypersurface X and so will not need to be recomputed. Furthermore, as of Magma V2.25, most of them now work for surfaces in ordinary projective ambients of dimension ≥ 4 with blow-up desingularisation data.

HomAdjoints(m, n, S): RngIntElt, RngIntElt, Srfc -> SeqEnum

```
UseFormalDesing: BoolElt                    Default: false
Classify       : Verbose                    Default: 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 $\mathbb{F}_{S,m}$ (*i.e.*, $\Gamma(S, \mathcal{O}_S(n) \otimes \mathbb{F}_{S,m})$) 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).

For S in $\mathbf{P}^r$, $r \geq 4$, with only point singularities this will also work, although the common degree d of the homogeneous polynomials in the sequence returned has no simple description. The sequence still defines the (m, n) adjoint-map on S and its length is equal to the dimension of global sections $H^0(X, \omega_X^{\otimes m}(n))$ where X is the desingularisation of S.

The intrinsic needs desingularisation data to work with and can use either formal desingularisation data or blow-up desingularisation data. It will first try to use desingularisation data (of either type) cached on the surface. If there is no cached precomputed data, it will call *ResolveSingularSurface* at the beginning to get such data (which will then have been cached with X). Use/generation of blow-up desingularisation data is now the default when S has only point singularities. To force the use of formal desingularisation data, the `UseFormalDesing` parameter should be set to `true`.

When using formal data, the intrinsic 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].

When using blow-up data, the same linear conditions are computed with the `LinearSystemDivisorRestriction` intrinsic and either the differential multiplicities intrinsic for hypersurfaces or the divisor of the base meromorphic 2-form computed in `BasisOfHolomorphicTwoForms` for surfaces in $\mathbf{P}^r$, $r \geq 4$. These are all described in section *Desingularisation by Blow Up*.

`GeometricGenusOfDesingularization(S): Srfc -> RngIntElt`

UseFormalDesing: BoolElt	Default: false

Given a hypersurface S in $\mathbf{P}^3$ or surface S with only point singularities in $\mathbf{P}^n$, $n \geq 4$, the intrinsic returns the geometric genus of (any) desingularization of S. The function just computes the dimension of the $(1,0)$ adjoint space. The parameter is the same as for `HomAdjoints`.

`PlurigenusOfDesingularization(S, m): Srfc, RngIntElt -> RngIntElt`

UseFormalDesing: BoolElt	Default: false

Given a hypersurface S in $\mathbf{P}^3$ or surface S with only point singularities in $\mathbf{P}^n$, $n \geq 4$, 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.The parameter is the same as for `HomAdjoints`.

`ArithmeticGenusOfDesingularization(S): Srfc -> RngIntElt`

UseFormalDesing: BoolElt	Default: false

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 coming from the Riemann–Roch theorem, involving the dimension of the (1,1)-adjoints and the degree of S. The parameter is the same as for `HomAdjoints`.

Example: Adjoint Spaces of a Surface `(ex-1ad4d2)`

We compute several adjoint spaces of a surface. We compute the resolution at the start, after which it is cached for the `HomAdjoints` calls. The resolution computes a formal desingularisation since the surface has a singular subscheme of dimension 1.

```
> P<x,y,z,w> := ProjectiveSpace(Rationals(), 3);
> F := w^3*y^2*z+(x*z+w^2)^3;
> S := Surface(P,F);
> desing := ResolveSingularSurface(S : AdjComp := true);
> #desing;
18
> HomAdjoints(1, 0, S);
[]
> HomAdjoints(1, 1, S);
[
    x*z*w + w^3
]
> HomAdjoints(1, 2, S);
[
    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);
[
    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);
[]
```

(continues on next page)

(continued from previous page)

```
> HomAdjoints(2, 1, S);
[]
> HomAdjoints(2, 2, S);
[
    x^2*z^2*w^2 + 2*x*z*w^4 + w^6
]
> HomAdjoints(2, 3, S);
[
    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
]
```

126.5.5 Classification and Parameterization of Rational Surfaces

This section contains intrinsics 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.

The intrinsic to test whether a surface is rational actually also applies to surfaces in higher-dimensional ordinary projective spaces as long as the surface model has no worse than simple singularities.

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 *Minimal Models*).

`IsRational(X): Srfc -> BoolElt`

```
UseFormalDesing: BoolElt                    Default: false
KnownADE       : 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 $\mathbf{P}^3$, there is no assumption about the singularity of X and a formal or blow-up desingularisation will be used to compute plurigenera, as long as either the characteristic of the base field is 0 or the singular subscheme is of dimension 0. If the ambient is $\mathbf{P}^r$, $r \geq 4$, then the intrinsic can use a blow-up desingularisation if X has singular subscheme of dimension 0.

If X has at worst simple (A-D-E) singularities, the generally faster sheaf algorithms of the first section may be used for the computations. The user can allow this method to be applied by setting the parameter `KnownADE` to `true` if they know this to be the case. Otherwise, simple singularities are not checked for and the sheaf method is not tried.

If a desingularisation first needs to be computed (if one is not already cached with X), as usual, blow-up desingularisation is the default (when it can be computed). To force the use of formal desingularisation in the hypersurface case, parameter `UseFormalDesing` should be set to `true`.

126.5.6 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:

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): Srfc -> Srfc, List, MonStgElt`

```
UseFormalDesing: BoolElt                    Default: false
Classify       : Verbose                    Default: 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.

If formal or blow-up desingularisation data has not already been cached with S, it is computed at the start. Parameter `UseFormalDesing` is as for earlier intrinsics.

Example: Classifying Rational Surfaces `(ex-f22fab)`

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,
```

(continues on next page)

(continued from previous page)

```
$.2*$.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 eighth 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
```

126.5.7 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): Srfc, Prj -> BoolElt, MapSch

```
UseFormalDesing: BoolElt                    Default: false
Classify       : Verbose                    Default: 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 intrinsic begins by mapping X, as in *ClassifyRationalSurface*, to a surface of special type. Then specialised parametrization routines are called depending on the type.

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.

As usual, if formal or blow-up desingularisation data has not already been cached with S, it is computed at the start. Parameter `UseFormalDesing` is as for earlier intrinsics.

`ParametrizeProjectiveSurface(X, P2): Srfc, Prj -> BoolElt, MapSch`

```
Classify: Verbose                    Default: 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: Parametrizing Rational Hypersurfaces (ex-83bd7e)

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
```

(continues on next page)

(continued from previous page)

```
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): RngMPolElt, FldFunRat -> SeqEnum

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: Parametrizing Factors of an Affine Hypersurface (ex-e08515)

The following affine hypersurface has three irreducible factors: one not rational and two that are rational and parameterizable over $\mathbf{Q}$.

```
> 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)
    ]
]
```

126.5.8 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): Sch, Prj -> BoolElt, MapSch`

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 $\mathbf{Q}$.

Example: Parametrizing Quadric Surfaces (ex-81f100)

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
```

(continues on next page)

(continued from previous page)

```
>                   - 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): MapSch, Prj -> BoolElt, MapSch`

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 [Sch00a]. The scheme X is not assumed to be non-singular.

Example: Parametrizing a Ruled Surface (ex-8ec1f8)

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): Sch, Prj -> BoolElt, MapSch`

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 anti-canonical 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 *Parametrization of Del Pezzo Surfaces* (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 *Parametrization of Del Pezzo Surfaces*.

126.6 Del Pezzo Surfaces

126.6.1 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`).

126.6.2 Creation of General Del Pezzos

`DelPezzoSurface(P, L): Prj, List -> SrfDelPezzo`

`DelPezzoSurface(S): Setq[Pt] -> SrfDelPezzo`

`DelPezzoSurface(Z): Clstr -> SrfDelPezzo`

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): RngMPolElt -> SrfDelPezzo`

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 $\mathbf{P}^3 = \text{Proj(P)}$. If this surface is not smooth an error is reported.

`IsDelPezzo(Y): Sch -> BoolElt, SrfDelPezzo, MapSch`

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.

126.6.3 Some Auxiliary Intrinsics

`CollinearPointsOnPlane(P2, S): Prj, SetIndx[Pt] -> SeqEnum`

Argument $P2$ should be an ordinary projective plane over a field and S an indexed set of distinct points on $P2$. Returns a sequence containing all subsets (as enumerated sequences) of 3 or more points of S that are collinear.

`PointsInGeneralPosition(P2, S): Prj, SetIndx[Pt] -> BoolElt, SeqEnum, SeqEnum, SeqEnum`

`NonGenPts: BoolElt`	Default: `false`

Argument $P2$ should be an ordinary projective plane over a field and S an indexed set of 8 or fewer distinct points on $P2$. Returns whether the points in S are in general position. This means that no 3 are collinear, no 6 lie on a conic and, if there are 8 points, they do not lie on a cubic with a double point at one of them.

If the `NonGenPts` parameter is set to `true` and the points are not in general position, also returns 3 sequences containing respectively the subsets of 3 or more collinear points, the subsets of 6 or more co-conic points and the full set of 8 points if they fail to satisfy the cubic condition (all as enumerated sequences).

`EckardtPoints(X): SrfDelPezzo -> Sch`

Argument X here should be a degree 3 Del Pezzo surface. Computes the set of Eckardt points of X, which are points where 3 coplanar lines of the surface meet. The result is returned as a zero-dimensional (or empty) scheme consisting of this set of points.

126.6.4 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 $1 \leq d \leq 9$. For $d \geq 3$, 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 $d \geq 3$, 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): MonStgElt, RngIntElt`

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): Sch -> BoolElt, MapIsoSch`

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.

In V2.19, LLL reductions have been applied to give simpler bases for the initial Lie algebra and the corresponding associative algebra. Also, the algorithm for splitting the latter has been changed to the one used with degree 3 homogeneous spaces of elliptic curves. This has greatly speeded up the computation.

`ParametrizeDegree8DelPezzo(X): Sch -> BoolElt, MapSch`

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 ${\bf P}^2$ 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: Parametrizing an Anticanonical Sphere (ex-2e5909)

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
```

(continues on next page)

(continued from previous page)

```
-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): Sch -> MapIsoSch

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): Sch -> BoolElt, MapIsoSch

```
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): FldNum, Pt -> Sch

Degree6DelPezzoType2_2(K, pt): FldNum, Pt -> Sch

Degree6DelPezzoType2_3(K, pt): FldNum, Pt -> Sch

Degree6DelPezzoType3(K, pt): FldNum, Pt -> Sch

Degree6DelPezzoType4(K, K1, pt): FldNum, Fld, Pt -> Sch

Degree6DelPezzoType6(K, pt): FldNum, Pt -> Sch

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): Sch -> BoolElt, MapIsoSch`

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: Parametrizing a Degree 6 Del Pezzo Surface `(ex-cfe2ed)`

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 defining 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);
```

(continues on next page)

(continued from previous page)

```
> 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 Gröbner 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 +
```

(continues on next page)

(continued from previous page)

```
    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): Sch -> MapIsoSch

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): Sch, Prj -> BoolElt, MapIsoSch

ParametrizeSingularDegree4DelPezzo(X, P2): Sch, Prj -> BoolElt, MapIsoSch

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: Parametrizing a Singular Cubic Del Pezzo (ex-b772c7)

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);
```

(continues on next page)

(continued from previous page)

```
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
```

126.6.5 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 Minimisation and Reduction) and plane quartics (see Section *Minimization and Reduction for Plane Curves*).

MinimizeCubicSurface(f, p): RngMPolElt, RngIntElt -> RngMPolElt, Mtrx

```
MinRedCubSurf: Verbose                    Default: 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): RngMPolElt -> RngMPolElt, Mtrx

```
MinRedCubSurf: Verbose                    Default: 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): MPolElt -> RngMPolElt, Mtrx

```
MinRedCubSurf: Verbose                    Default: 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): SeqEnum, RngIntElt -> SeqEnum, Mtrx`

```
MinRedDeg4delPezzo: Verbose                    Default: 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): SeqEnum -> SeqEnum, Mtrx`

```
MinRedDeg4delPezzo: Verbose                    Default: 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 $\mathbf{Q}$ scheme as that defined by the returned quadrics.

For the reduction step, `ReduceQuadrics` is called.

`MinimizeReduce(S): SrfDelPezzo -> SrfDelPezzo, Mtrx`

```
MinRedCubSurf     : Verbose                    Default: 2
MinRedDeg4delPezzo: Verbose                    Default: 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: Minimizing Cubic and Quartic Del Pezzo Surfaces (ex-0d6a36)

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 -
```

(continues on next page)

(continued from previous page)

```
    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
```

126.6.6 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): RngMPolElt -> RngIntElt, RngIntElt

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: Counting Points on a Cubic Surface (ex-3f2682)

```
> 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);
265613988875874769338781322035779626829233452805946242111250315725884985311926O\\
    79714208525578202
13
```

So we get a large number of points and the Frobenius has Swinnerton-Dyer number 13.

IsIsomorphicCubicSurface(f, g): MPolElt, MPolElt -> BoolElt, List

```
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: Isomorphism Testing of Cubic Surfaces (ex-5e111c)

```
> _<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.

126.6.7 Construction of Cubic Surfaces

`CubicSurfaceByHexahedralCoefficients(p): RngUPolElt -> RngMPolElt`

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 [EJ10a] for details.

`CoblesRadicand(p): RngUPolElt -> FldElt`

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: Cubic Surface from Hexahedral Coefficients (ex-f2bff8)

```
> 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)]);
13021618703131414094337256000 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));
```

(continues on next page)

(continued from previous page)

```
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.

126.6.8 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.

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): RngMPolElt -> SeqEnum, RngElt`

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): RngMPolElt -> RngElt`

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): SeqEnum -> RngMPolElt`

Computes a cubic surface whose invariants are equal to the given sequence. The algorithm requires the last invariant to be non-zero.

Example: Invariants of a Cubic Surface (ex-7e1f83)

```
> 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, 19060725633060987807534362396227812500/9,
9303881097343297830094619181364801014510415259433593750/9,
3633112616588281944217451032208493848831995668840667046827293985351562500/81,
443346812297707471151312880259534705807785333028016737274243674227613642460\\
3575851440429687 5/243 ]
> [inv[i] / inv[1]^i : i in [1..5]];
[ 1, 2, 3, 4, 5 ]
> SkewInvariant100(surf);
-14747658751687702477522103639772055954980186626723314226839431502066804810\\
8692101231381870506424535465849889285142677691430466259169068950411766128 2\\
5810551009923477912337277997297305679991266945261596766795264557003974914 5\\
50781250/531441
```

Thus the constructed surface has equivalent invariants (as they are in $\mathbf{P}(1,2,3,4,5)$) and no Eckardt points.

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): RngMPolElt -> SeqEnum`

The intrinsic constructs a sequence containing Salmon's 4 linear covariants for the cubic surface f.

`ClassicalCovariantsOfCubicSurface(f): RngMPolElt -> SeqEnum`

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.

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): RngMPolElt, UserProgram, SeqEnum -> RngElt`

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): RngMPolElt -> SeqEnum`

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: Dual of the Cayley Cubic (ex-b242ba)

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>
]
```

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): MPolElt, MPolElt -> MPolElt`

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: Salmon's First Invariant (ex-b667d5)

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
```

126.6.9 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): RngMPolElt -> RngMPol`

Computes the ideal of the faces of the pentahedron of the cubic surface f.

Example: Pentahedron of a Random Cubic Surface (ex-aa479a)

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) @}
```

126.7 Degree 2 K3 Surfaces

126.7.1 Introduction

We refer to K3 surfaces that can be represented as the desingularisation of a double cover of the projective plane $\mathbf{P}^2$ ramified over a sextic curve as *degree* 2 K3 surfaces. This condition on K3 surfaces is equivalent to the existence of an irreducible divisor of self-intersection 2. The divisor map of such a divisor is the required double cover of $\mathbf{P}^2$. Conversely, the pullback of a general line in a double-cover presentation gives an irreducible divisor of self-intersection 2.

If we take a (normal) double cover of the plane ramified over a reduced sextic (degree 6) curve, the desingularisation gives a K3 surface if and only if the curve has at worst simple curve singularities (s-c-s). If the singularities are worse, the resulting double-cover surface has Kodaira dimension -1. Given the s-c-s condition, the normal double cover ramified over the sextic has simple singularities precisely over the singularities of the sextic (so is already non-singular if the sextic is). For more information on this and more general surface double covers, see Ch. III, Section 7 and Ch. V, Section 22 of [BHPdV04].

This section describes functionality to work with degree 2 K3 surfaces given by the (generally singular) normal double-cover model naturally embedded in a weighted $\mathbf{P}(1,1,1,3)$ space (see the basic constructor below). Singularities and the set of (-2)-curves lying over them are handled by blow-up desingularisation. The only restriction on the base-field is that the characteristic cannot be equal to 2.

The current functionality mainly deals with computation of (-2)-curves coming from lines and conics that split in the double cover (along with those above singular points), the intersection matrix of a given finite collection of curves and blow-up divisors on the surface, and elliptic fibrations of the surface related to appropriate (-2)-curve configurations. We hope to expand this functionality in a number of directions in future releases.

126.7.2 Creation Functions

Currently, there is no special surface subtype for degree 2 K3 surfaces, but we construct them in a standard form (in weighted $\mathbf{P}(1,1,1,3)$ ambients) which the intrinsics in this section work with. The main constructor below takes a homogeneous degree 6 polynomial f in 3 variables, checks that it gives a reduced plane curve with only simple curve singularities, and returns this standard form for the K3 surface which is the double cover of the plane ramified over f. The singular subscheme of the surface is also filled in.

Note that later intrinsics assume that singularity properties of the surface argument have been computed and cached, so this constructor should always be used to create degree 2 K3 surfaces. The `SingularSubscheme` intrinsic currently doesn't work directly for these surfaces because of the weighted ambient.

`DegreeTwoK3Surface(f): RngMPolElt -> Srfc`

The argument f should be a homogeneous degree 6 polynomial in a 3-variable homogeneous polynomial ring over a field of characteristic $\neq 2$. Returns the surface S, which is a (generally singular) model of the K3 surface that is a double cover of the projective plane ramified over the $f = 0$ curve C_f.

C_f should be a reduced curve with at worst simple curve singularities. This is checked, with an error resulting if the conditions are not satisfied. Simple curve singularities are double and triple points analytically equivalent to particular forms as listed in Thm. 8.1 of Chapter II, [BHPdV04]. They lie in the usual A-D-E families and a curve singularity of a certain type on C_f gives a simple surface singularity of the same type on S (e.g. $A_2 \to A_2$) lying over it. Note that C_f does not have to be irreducible. If, for example, it has two components meeting transversely at a point,

that point is a simple A_1 singularity of C_f (node). Nodes are A_1 singularities and simple cusps A_2 singularities on a curve.

If f is defined over the base field k then S will be a hypersurface in a weighted projective space with variables x, y, z, w which have weights $1, 1, 1, 3$ respectively. The defining polynomial will be $f(x, y, z) - w^2$. S is a normal variety giving the normal double cover of the plane with ramification locus C_f. As mentioned above, this will have simple singularities over singularities of C_f and the actual K3 surface is the desingularisation of S.

The intrinsic also computes and caches the singular subscheme of S, if it is singular, or caches that it is non-singular otherwise, before it is returned.

Strictly speaking, the double cover of the plane ramified over C_f is really only defined as a variety over k up to quadratic twist for a non algebraically-closed base field k. The particular twist constructed is determined by f in choosing the defining polynomial $f - w^2$.

Example: Constructing the Drell-Yan K3 Surface (ex-25b20a)

We construct the *Drell-Yan* K3 surface, which is related to certain Feynmann integrals arising in high-energy physics (see [BFHN]). The ramification locus for this interesting surface has many singularities which give rise to 14 blow-up (-2)-curves via desingularisation. We will use this example again later to illustrate the various intrinsics.

```
> R<a,b,c> := PolynomialRing(Rationals(),3);
> f := 4*a*b^2*c*(a+c)^2+(a*(b+c)^2+b*(a+c)^2)*(a*(-b+c)^2+b*(c-a)^2);
> S := DegreeTwoK3Surface(f);
> S; Ambient(S);
Surface over Rational Field defined by
x^4*y^2 + 2*x^3*y^3 + x^2*y^4 + 4*x^3*y^2*z + 2*x^3*y*z^2 - 4*x^2*y^2*z^2 +
2*x*y^3*z^2 + 4*x*y^2*z^3 + x^2*z^4 + 2*x*y*z^4 + y^2*z^4 - w^2
Projective Space of dimension 3 over Rational Field
Variables: x, y, z, w
The grading is:
    1, 1, 1, 3
> MinimalBasis(SingularSubscheme(S));
[
    x*y + x*z,
    w,
    y^2*z + y*z^2,
    x^2*z + y*z^2
]

> HasOnlySimpleSingularities(S : ReturnList := true);
true
[* <(-1 : -1 : 1 : 0), "A", 1>, <(1 : -1 : 1 : 0), "A", 3>,
 <(0 : 0 : 1 : 0), "A", 2>, <(0 : 1 : 0 : 0), "A", 4>,
 <(1 : 0 : 0 : 0), "A", 4> *]
```

126.7.3 Split Divisors and Intersection Matrices

One of the main aims of the analysis of a K3 surface S is to compute its Neron-Severi group $NS(S)$, its divisor group up to algebraic equivalence. For a K3, this is a finitely-generated, torsion-free abelian group of rank ≤ 20 (for S in characteristic 0) and algebraic equivalence is the same as rational equivalence or numerical equivalence so $NS(S) = Pic(S) = Num(S)$.

In many cases, $NS(S)$ is generated by (-2)-curves. Apart from the irreducible blow-up divisors over singular points, one way to find (-2)-curves on a degree 2 K3 surface is as the components of the pullbacks of lines and conics in the plane which split in the double covering. This idea was first exploited by Elsenhans and Jahnel and formulated into an algorithm by Dino Festi ([Fes]).

We provide an intrinsic to compute the split lines and also some prototype intrinsics to compute split

conics. We hope to add efficient intrinsics to properly compute the split conics for characteristic 0 degree 2 K3 surfaces in subsequent releases. As it stands, we provide an intrinsic to compute split conics when S is defined over a (small) finite field. When applied to reductions mod p for several p, this often provides split conics that can be lifted to $\mathbf{Q}$ (see the example). There is also an intrinsic to check whether a given conic splits or not.

It should be stated that these intrinsics tend to find split curves in the high-rank cases ($rk(NS(S))$ is close to 20) but often return nothing for lower-rank cases, where constructing non-trivial divisor classes is still a hard problem.

These split divisor intrinsics were adapted from code kindly provided by D. Festi.

We also provide an intrinsic that computes the full intersection matrix of the set consisting of the strict transforms of a given collection of effective divisors on S and the irreducible blow-up divisors over singularities of the ramification sextic. This, together with the intrinsics to find split curve divisors, allows the partial, or in some cases complete, computation of $NS(S)$.

`TriTangentLines(S): Srfc -> List`

`TriTangentLines(F): RngMPolElt -> List`

This computes and returns a list of the full set of lines in the plane that split in the degree 2 K3 surface S. Since this condition is equivalent to the line meeting the ramification sextic with even multiplicity at each intersection point, we refer to these as tri-tangent lines. Some may be defined over a finite extension of the base field k of S, in which case only one of the set of Galois-conjugate lines is returned.

There are two versions, one taking the surface S as argument and the other taking the degree 6 ramification polynomial F, which should lie in a 3-variable polynomial ring. In the first case, the lines are returned as linear forms in the first 3 variables of the coordinate ring of the ambient of S, R, and lie in a base change of R to the field of definition of the line. In the second case, the return values are linear forms in a base change of the parent polynomial ring of F.

`SixTangentConicsModp(F): RngMPolElt -> SeqEnum[RngMPolElt]`

This computes the full set of conics that split in a degree 2 K3 surface S defined over a finite field k. Since it is largely intended for usage on the mod p reduction of surfaces defined over $\mathbf{Q}$ (or a finite extension of $\mathbf{Q}$), we have included "Modp" in the name and only include a version where the argument is a degree 6 homogeneous polynomial F in 3 variables defining the ramification locus of S as a double cover of the plane. As above, the conics that split are precisely those that intersect this sextic ramification curve with even multiplicity at each intersection point, so we refer to them as six-tangent conics.

As opposed to `TriTangentLines`, this only computes split conics which are defined over k and not over finite extensions. It uses an iteration through all possible conics, so overall speed of execution decreases very quickly with the size of k. Thus, it should only be used over quite small finite fields.

Note that a quadratic factor of F is not counted as a six tangent conic by this intrinsic, even though it ramifies in the double cover to give a (-2)-curve on S.

The return value is a sequence of homogeneous degree 2 polynomials in the parent polynomial ring of F.

`SixTangentConicCheck(S, C): Srfc, RngMPolElt -> BoolElt`

`SixTangentConicCheck(F, C): RngMPolElt, RngMPolElt -> BoolElt`

Returns whether a plane conic splits in the degree 2 surface S or not. There are two versions, one taking the surface S as its first argument and the second taking the homogeneous degree 6 ramification polynomial F, which should lie in a 3-variable polynomial ring which MUST have the grevlex ordering.

In either case the second argument is a homogeneous degree 2 polynomial C giving the conic equation. In the first case, it should be a polynomial in the first 3 variables of the coordinate ring of the ambient of S. In the second, it should lie in the same polynomial ring as F.

`IntersectionMatrixOnDegree2K3(S, Ds): Srfc, SeqEnum[Sch] -> Mtrx`

```
IncludeExCurves: BoolElt                    Default: true
```

Argument S should be a degree 2 K3 surface in standard form and argument Ds a sequence of effective divisors given as subschemes of S. No two of Ds should have an irreducible component in common. This is not checked, nor is whether the $D \in Ds$ actually form divisors (i.e. have a pure codimension 1 primary decomposition). The expected case is that the divisors are all distinct and irreducible, in which case there is no problem.

Let `seq` be the sequence of divisors given by appending any irreducible blow-up curves ("exceptional" curves) to the sequence of strict transforms on the desingularised surface of the divisors in Ds. The order in which they are appended is s_1 *cat* s_2 *cat* $\ldots s_n$, where s_i is the sequence of exceptional divisors for the i-th desingularisation data object $dsds[i]$ of the desingularisation data list $dsds$ returned by `ResolveSingularSurface`. The order of blow-up divisors in s_i is the order in which they are indexed in $dsds[i]$.

The intrinsic returns the #`seq`-by-#`seq` intersection matrix for the divisors in `seq` on the K3 surface.

To get the intersection matrix of the strict transforms without the appended exceptional curves, the `IncludeExCurves` parameter can be set to `false` (the default is `true`).

Example: Intersection Matrix of the Drell-Yan Surface `(ex-54e6f4)`

We construct (-2)-curves on the Drell-Yan surface S via split lines and conics and compute the rank of the subgroup of $NS(S)$ that they generate together with the 14 exceptional curves by using the intersection matrix intrinsic.

```
> R<a,b,c> := PolynomialRing(Rationals(),3);
> f := 4*a*b^2*c*(a+c)^2+(a*(b+c)^2+b*(a+c)^2)*(a*(-b+c)^2+b*(c-a)^2);
> S := DegreeTwoK3Surface(f);
> lins := TriTangentLines(S);
> lins;
[*
    x - z,
    x,
    y + z,
    y,
    y + alpha1*z,
    z
*]
```

Find which lines are defined over $\mathbf{Q}$.

```
> [IsIdentical(Parent(l),CoordinateRing(Ambient(S))) : l in lins];
[ true, true, true, true, false, true ]
> BaseRing(Parent(lins[5]));
Number Field with defining polynomial u^2 - 3*u + 1 over the Rational Field
```

The fifth line gives two conjugate split lines defined over a quadratic extension of $\mathbf{Q}$ ($\mathbf{Q}(\sqrt{5})$) and the others are defined over $\mathbf{Q}$. We ignore the fifth one here, to avoid base extension, and get the (-2)-curves over $\mathbf{Q}$ defined by the others.

```
> spl := [PrimeComponents(Scheme(S,lins[i])) : i in [1,2,3,4,6]];
> assert &and[#ps eq 2 : ps in spl];
> Ls := [s[1] : s in spl];
> Lsd := [s[2] : s in spl];
> Ls;
[
    Scheme over Rational Field defined by
    y^2*z + 2*y*z^2 + z^3 + w,
```

(continues on next page)

(continued from previous page)

```
    x - z,
    Scheme over Rational Field defined by
    y*z^2 + w,
    x,
    Scheme over Rational Field defined by
    x^2*z - z^3 + w,
    y + z,
    Scheme over Rational Field defined by
    x*z^2 + w,
    y,
    Scheme over Rational Field defined by
    x^2*y + x*y^2 + w,
    z
]
> Lsd;
[
    Scheme over Rational Field defined by
    y^2*z + 2*y*z^2 + z^3 - w,
    x - z,
    Scheme over Rational Field defined by
    y*z^2 - w,
    x,
    Scheme over Rational Field defined by
    x^2*z - z^3 - w,
    y + z,
    Scheme over Rational Field defined by
    x*z^2 - w,
    y,
    Scheme over Rational Field defined by
    x^2*y + x*y^2 - w,
    z
]
```

Now we see if we can find any split conics by searching mod 5 and mod 7.

```
> Rp<a,b,c> := PolynomialRing(GF(5),3,"grevlex");
> SixTangentConicsModp(Rp!f);
[
    a^2 + a*b + b^2 + a*c + 2*b*c,
    a^2 + 2*a*b + 2*a*c + c^2,
    4*a^2 + 4*a*b + b^2 + 2*a*c + 3*b*c,
    a^2 + b*c,
    a^2 + 2*a*b + 2*a*c + b*c
]
> Rp<a,b,c> := PolynomialRing(GF(7),3,"grevlex");
> SixTangentConicsModp(Rp!f);
[
    a^2 + a*b + b^2 + a*c + 2*b*c,
    a^2 + b*c
]
```

We suspect that the two mod 7 split conics may be split conics over $\mathbf{Q}$ with the natural lifted equations, since they both occur as mod 5 split conics. This is indeed true, as we now check.

```
> x,y,z,w := Explode([Ambient(S).i : i in [1..4]]);
> Cs := [x^2+y*z, x^2+x*y+y^2+x*z+2*y*z];
```

(continues on next page)

(continued from previous page)

```
> [SixTangentConicCheck(S,C) : C in Cs];
[ true, true ]
> splC := [PrimeComponents(Scheme(S,C)) : C in Cs];
> assert &and[#ps eq 2 : ps in splC];
> LsC := [s[1] : s in splC];
> LsCd := [s[2] : s in splC];
> [MinimalBasis(Z) : Z in LsC];
[
    [
        x^2 + y*z,
        x*y^2 - y^2*z - x*z^2 - y*z^2 - w
    ],
    [
        x^2 + x*y + y^2 + x*z + 2*y*z,
        y^3 + x*y*z + 4*y^2*z + x*z^2 + 3*y*z^2 + w
    ]
]
> [MinimalBasis(Z) : Z in LsCd];
[
    [
        x^2 + y*z,
        x*y^2 - y^2*z - x*z^2 - y*z^2 + w
    ],
    [
        x^2 + x*y + y^2 + x*z + 2*y*z,
        y^3 + x*y*z + 4*y^2*z + x*z^2 + 3*y*z^2 - w
    ]
]
```

We now get the intersection matrix for the $28 = 10 + 4 + 14$ (-2)-curves consisting of the split line and conic divisors and the 14 blow-up curves, all of which are defined over $\mathbf{Q}$.

```
> Ls cat:= LsC;
> Lsd cat:= LsCd;
> imat := IntersectionMatrixOnDegree2K3(S, Ls cat Lsd);
> imat;
[ -2 0 0 1 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 -2 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 ]
[ 0 1 -2 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 ]
[ 1 0 0 -2 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 ]
[ 0 0 0 0 -2 0 0 0 0 0 0 1 1 2 0 0 0 0 0 0 1 0 0 0 1 0 0 0 ]
[ 0 0 0 1 0 -2 0 0 0 0 0 1 2 1 1 0 1 0 1 0 0 1 0 0 0 0 0 0 ]
[ 1 0 0 1 0 0 -2 0 1 0 0 2 1 3 1 1 0 0 0 1 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 -2 0 0 1 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 1 0 -2 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 1 -2 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 ]
[ 0 0 0 0 0 0 0 1 0 0 -2 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 ]
[ 0 0 0 0 1 1 2 0 0 0 0 -2 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 ]
[ 0 0 0 0 1 2 1 0 0 0 1 0 -2 0 1 1 0 0 0 1 1 0 0 0 0 0 0 0 ]
[ 0 1 0 0 2 1 3 1 0 0 1 0 0 -2 1 0 1 0 1 0 0 0 0 0 0 0 0 0 ]
[ 0 0 1 0 0 1 1 0 0 1 0 0 1 1 -2 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 -2 0 1 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 -2 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 1 1 -2 0 0 0 0 0 0 0 0 0 0 ]
[ 0 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 -2 1 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 1 -2 0 0 0 0 0 0 0 0 ]
```

(continues on next page)

(continued from previous page)

```
[ 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 -2 0 1 0 0 0 0 0 ]
[ 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 -2 0 1 0 0 0 0 ]
[ 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 -2 1 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 -2 0 0 0 0 ]
[ 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 -2 0 1 0 ]
[ 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 -2 0 1 ]
[ 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 -2 1 ]
[ 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 -2 ]
> Rank(imat);
18
```

This shows that the 28 curves generate a subgroup of $NS(S)$ of rank 18. In fact, [BFHN] shows that the rank of the subgroup of $NS(S)$ defined over $\mathbf{Q}$ is 18, that its full rank (over $\bar{\mathbf{Q}}$) is 19 and that the full NS(S) is generated by the above (-2)-curves along with the divisors of the split line defined over $\mathbf{Q}(\sqrt{5})$ and those of a split conic also defined over $\mathbf{Q}(\sqrt{5})$. The above curves probably do generate the full subgroup defined over $\mathbf{Q}$ but I haven't checked this.

126.7.4 Elliptic Fibrations

Elliptic fibrations of K3 surfaces are everywhere defined maps from the surface to $\mathbf{P}^1$ such that the general fibre is an irreducible, non-singular genus 1 curve. Sometimes these are referred to as genus 1 fibrations with the elliptic fibration terminology reserved for the stronger condition that there also exists a section of the fibre map $\Leftrightarrow$ the generic genus 1 fibre has a point rational over its base-field. However, we do not distinguish here. When there is a section, uniformly providing a O-point on all fibres, the K3 surface with its fibre map actually gives the minimal (curve) model over $\mathbf{P}^1$ of the generic elliptic curve fibre.

Such fibrations are very useful in analysing arithmetic and geometric properties of a K3 surface S and finding multiple inequivalent fibrations can also be important, providing different decompositions of finite index sublattices of the Neron-Severi lattice and leading to automorphisms of S with interesting properties.

We use a method to compute elliptic fibrations based upon having many (-2)-curves on the surface at our disposal and finding configurations of these that give bad fibres of an elliptic fibration. This idea has appeared (in a more theoretical setting) previously in the works of Shioda. It was discovered independently as a computational tool by Michael Harrison.

We refer to the configurations of (-2)-curves that give elliptic fibrations as *Kodaira configurations*. These are sets of (-2)-curves that intersect in the right way to give a standard non-irreducible elliptic fibration bad fibre, when given the correct multiplicity. Each Kodaira configuration on a K3 surface is the bad fibre of a unique (up to automorphisms of the base $\mathbf{P}^1$) elliptic fibration and two Kodaira configurations give the same fibration if and only if they are disjoint (see Section 4 of [BFHN]).

The elliptic fibration intrinsics in this section provide the following functionality. Given the intersection matrix of a collection, Σ, of (-2)-curves on S, there is an intrinsic that computes all of the maximal disjoint sets of Kodaira configurations supported on Σ. For a particular Kodaira configuration, consisting of (-2)-curves that may be either the strict transforms of curves on the singular model of S or irreducible blow-up curves over the simple singularities, there is an intrinsic to explicitly compute the associated elliptic fibration map to $\mathbf{P}^1$. Additionally, there is an intrinsic to compute the generic fibre, as a curve over a rational function field $k(t)$ where k is the base field of S, of an elliptic fibration from its map. This intrinsic also computes the associated $k(t)$-rational points on the generic fibre for a given set of sections of the fibration.

KodairaConfigurations(imat): Mtrx -> List

```
KodFibs: Verbose                    Default: 1
```

The argument *imat* should be the intersection matrix of an ordered set Σ of irreducible (-2)-curves on the desingularisation of a degree 2 K3 surface S. Returns a list containing all of the maximal

disjoint sets of Kodaira configurations supported on Σ (see the introductory section). The list is in 1-1 correspondence with equivalence classes of elliptic fibrations on S that have at least one bad fibre supported entirely on curves in Σ. The set of Kodaira configurations in the list entry gives all of the bad fibres of the corresponding fibration that are supported on curves in Σ. Here, we say that two elliptic fibration maps to $\mathbf{P}^1$ are equivalent if they simply differ by an automorphism of $\mathbf{P}^1$ and that two elliptic fibrations are equivalent if the associated equivalence classes of elliptic fibration maps are the same. Equivalent elliptic fibrations clearly have the same set of fibres.

The list entry for a maximal disjoint set of configurations is a sequence of the form $[e_1, \ldots, e_r]$ where the e_i are the configurations given in the form of a pair $\langle s, l \rangle$ where s is the a sequence of indices of the elements of Σ forming the components of the fibre and l is a label string giving the type of the fibre.

The type labels use the extended Dynkin notation for elliptic bad fibres, except that I_n is used rather than $\tilde{A}_n$ for multiplicative reduction types. The code cannot distinguish between types III and I_2 or IV and I_3 (this doesn't matter in practice: the fibres contain each irreducible component to multiplicity 1 either way), and the I_n label will be used in both cases. The possible type labels are thus I_n, $n \geq 2$, D_n, $n \geq 4$, and E_6, E_7, E_8, where the D_i and E_i types are usually written with a tilde and represent a configuration of (-2)-curves forming an *extended* Dynkin diagram of the type label in terms of their dual intersection graph. See Ch. V, Sec. 7 of [BHPdV04] for a description of these configurations.

To know what multiplicities of the components of the fibre occur, it is necessary to know which nodes of the corresponding extended Dynkin diagram correspond to which positions in the sequence s. This depends on the type of the configuration.

For an I_n configuration, whose diagram is a (regular) n-gon, the order of s gives the cyclic order to which the components occur. That is, component $s[1]$ is joined to component $s[2]$, component $s[2]$ to component $s[3]$, ... component $s[n-1]$ to component $s[n]$, and component $s[n]$ to component $s[1]$.

For an extended D_n configuration, if $s = [a_1, .., a_{n+1}]$ then the central axis of the extended Dynkin diagram is $a_3 - a_4 - \ldots - a_{n-1}$ and a_1, a_2 join onto the a_3 node, a_n and a_{n+1} join onto the a_{n-1} node [a_3 is the central node when $n = 4$].

For an extended E_6 configuration, $s = [a_1, \ldots, a_7]$ and in the extended Dynkin diagram, a_1 is the central node and $a_2 - a_3$, $a_4 - a_5$, $a_6 - a_7$ are the arms with a_2, a_4, and a_6 meeting a_1.

For an extended E_7 configuration, $s = [a_1, \ldots, a_8]$ and in the extended Dynkin diagram, $a_1 - a_2 - \ldots - a_6 - a_7$ is the main axis with a_8 meeting a_4.

For an extended E_8 configuration, $s = [a_1, \ldots, a_9]$ and in the extended Dynkin diagram, $a_1 - a_2 - \ldots - a_7 - a_8$ is the main axis with a_9 meeting a_3.

Example: Elliptic Fibrations of the Drell-Yan Surface (ex-b2ea2c)

We continue with the Drell-Yan surface, using the 28 (-2)-curves from the last example as our Σ and the intersection matrix computed there. We find a lot of inequivalent elliptic fibrations.

```
> SetVerbose("KodFibs",1);
> time confs := KodairaConfigurations(imat);
Getting simple paths...
Time: 2.590
Getting In loops...
Time: 0.610
Getting extended Dn configs...
Time: 1.540
Getting extended E6 configs...
Time: 0.070
Getting extended E7 configs...
Time: 0.070
Getting extended E8 configs...
```

(continues on next page)

(continued from previous page)

```
Time: 0.190
Getting disjoint sets of configurations...
Dealing with 16716 configurations.
Done 1000 sets of configurations
Done 2000 sets of configurations
Done 3000 sets of configurations
Done 4000 sets of configurations
Done 5000 sets of configurations
Done 6000 sets of configurations
Done 7000 sets of configurations
Done 8000 sets of configurations
Done 9000 sets of configurations
Done 10000 sets of configurations
Done 11000 sets of configurations
Done 12000 sets of configurations
Done 13001 sets of configurations
Done 14001 sets of configurations
Done 15001 sets of configurations
Done 16001 sets of configurations
Time: 11.460
> #confs;
16402
```

We have found 16402 inequivalent elliptic fibrations of S! We examine the configuration data for some of them.

```
> Max([#e : e in confs]);
3 2022
> // 3 is the maximum number of disjoint Kodaira fibrations in
> // one of the configuration sets.
> // Get the indices in confs of the size 2 and 3 configuration sets
> s2 := [i : i in [1..#confs] | #(confs[i]) eq 2];
> #s2;
282
> s3 := [i : i in [1..#confs] | #(confs[i]) eq 3];
> #s3;
16
> confs[s3[3]];
[
    <[ 16, 18, 1, 4, 27, 28, 26, 3 ], "I8">,
    <[ 10, 20, 9, 24, 22, 23 ], "D5">,
    <[ 5, 14 ], "I2">
]
```

The elliptic fibration corresponding to the above configuration has 3 bad fibres which are supported solely on curves in Σ. They are multiplicative reduction fibres of type I_8 and I_2 and an additive reduction fibre of extended Dynkin type $\tilde{D}_5$ (I_1^* in the alternative bad fibre notation). Using the order of indexing of our (-2)-curves and the notation $E_{i,j}$ for the jth blow-up divisor of the ith desingularisation data object $dsds[i]$ of S, we see that the additive bad fibre as a divisor on the desingularisation of S is $L_3^d + E_{3,2} + E_{4,2} + E_{4,3} + 2 * L_2^d + 2 * E_{4,4}$ where L_i^d is the strict transform of the curve `Lsd[i]`.

The 1968th elliptic fibration will be constructed in the next example. It's configuration data contains two bad fibres supported on Σ: an I_{10} and an I_2 fibre.

```
> confs[1968];
[
```

(continues on next page)

(continued from previous page)

```
    <[ 25, 10, 9, 24, 23, 2, 3, 26, 28, 27 ], "I10">,
    <[ 6, 13 ], "I2">
]
```

EllipticFibrationRRSpaceDeg2K3(S, divlst, exdivlst): Srfc, SeqEnum, SeqEnum -> SeqEnum[RngMPolElt], RngMPolElt

The first argument S should be a degree 2 K3 surface in standard form. Given a Kodaira fibre F on the desingularisation X of S, the intrinsic returns the 2-dimensional Riemann–Roch space of F, $H^0(X, L(F))$, as a sequence B and denominator d. Here $B = [f_1, f_2]$ where f_1, f_2, d are all weighted homogeneous polynomials of the same degree in the coordinate ring of the ambient of S, and a basis for the R-R space is given by the two rational functions on S, f_1/d and f_2/d.

The return is in this form for convenience because the divisor map from $S \to \mathbf{P}^1$ of $|F|$, which is the elliptic fibration map with F as a bad fibre, is then just defined by B.

The components of F may be strict transforms of irreducible curves on S or irreducible blow-up divisors over the singular points of S. F is specified by the second and third argument as follows. Assume that, as a divisor on X, it is given by the sum $m_1 * D_1 + \ldots + m_r * D_r + n_1 * E_1 + \ldots + n_s * E_s$, where the D_i are the irreducible components of F that are strict transforms of curves on S, the E_i are irreducible blow-up divisors and the m_i, n_j are positive multiplicities.

The second argument `divlst` specifies the (always non-empty) D_i part of the divisor sum. It should be a sequence of (`Sch`,`RngIntElt`) 2-element lists where the ith 2-element list is $[*Z_i, m_i*]$ where Z_i is the subscheme of S whose strict transform is D_i.

The third argument `exdivlst` specifies the E_i part of the sum. It should be a sequence of (`Sch`, `RngIntElt`, `RngIntElt`) 3-element lists. For the $n_i * E_i$ term of the sum, if E_i is the jth irreducible divisor over the singularity represented by desingularisation data object `dsd`, the corresponding 3-element list would be $[*dsd, j, n_i*]$. If there are no E_i, this sequence should just be the empty sequence.

A concrete example of using this intrinsic is given below.

The computation uses a simplified version of the standard Magma Riemann–Roch functionality applied to an ordinary projective embedding of S into ordinary $\mathbf{P}^{10}$, with simplifications coming from the fact that this is a projectively-normal embedding. The blow-up divisor part is handled by the linear system divisor restriction intrinsic.

EllipticGeneralFibreDeg2K3(S, B, secs): Srfc, SeqEnum[RngMPolElt], SeqEnum[Sch] -> Crv, SeqEnum, SeqEnum

```
varname: MonStgElt                    Default: "t"
```

As usual, the first argument S should be a degree 2 K3 surface in standard form. The second argument B should be a sequence of two weighted-homogeneous polynomials of the same degree in the coordinate ring of the ambient of S that define an elliptic fibration map on S. E.g., as returned by the previous intrinsic. The third argument `secs` should be a (possibly empty) sequence of irreducible subschemes of S whose strict transforms to the desingularisation of S give sections of this fibration. If F is any fibre of the elliptic fibration (e.g. a Kodaira fibre defining it), then the condition that the strict transform $\tilde{D}$ of irreducible $D \subset S$ is a section is precisely that the intersection number $\tilde{D}.F = 1$. This is easily checked with the intersection matrix intrinsic.

If the base field of S is k, the intrinsic returns a (generally singular) curve C over rational function field $k(t)$ that is a birational model of the smooth genus 1 generic fibre of the fibration map. Technically speaking, this is the scheme-theoretic pullback of the fibration $S \to \mathbf{P}^1$ under the generic point inclusion map $Spec(k(t)) \to \mathbf{P}^1$. t corresponds to the rational function $B[1]/B[2]$ on S. C will usually be returned as a plane curve.

A second return value gives the sequence of $k(t)$-rational points on C corresponding to the sections in `secs`.

The third return value is a sequence of 4 rational functions on the ambient of C that give the substitutions for x, y, z, w (the coordinate variables on the ambient of S) that transform S generically into C.

By default, the variable generator t of the rational function field $k(t)$ over which C is defined, will be labelled "t". To change this, the `varname` parameter can be used to specify the string used for the variable labelling.

Example: Weierstrass Model of an Elliptic Fibration `(ex-d5a79f)`

We construct one of the elliptic fibrations of the Drell-Yan surface coming from the configurations found in the last example, compute its generic fibre, and use a point on this fibre coming from a section to transform to a Weierstrass elliptic curve model over $\mathbf{Q}(t)$.

The fibration we will construct is the one corresponding to the 1968th configuration found, $confs[1968]$. It has two bad fibres supported on our set of 28 curves, an I_2 and an I_{10} fibre. Although it is simpler to construct the fibration from the I_2 fibre, we will use the I_{10} fibre, which contains blow-up divisors, to better illustrate the Kodaira fibre divisor specification.

In the notation of the last example, the I_{10} Kodaira fibre is

$$L_2 + L_2^d + L_3 + L_3^d + E_{4,3} + E_{4,4} + E_{5,1} + E_{5,2} + E_{5,3} + E_{5,4}$$

We also note that L_4, L_4^d, L_5, L_5^d each give sections of the fibration as is easily deduced from the intersection matrix $imat$.

```
> dsds := ResolveSingularSurface(S); // get the desing data objects
> B,F := EllipticFibrationRRSpaceDeg2K3(S,[ [*Ls[2],1*],[*Lsd[2],1*],
> [*Ls[3],1*], [*Lsd[3],1*] ], [ [*dsds[4],j,1*] : j in [3,4] ] cat
> [ [*dsds[5],j,1*] : j in [1..4] ]);
> B; F;
[
    x^2 + y*z,
    x*y + x*z
]
x*y + x*z
```

Now we get the generic fibre of the fibration and the points on it corresponding to the sections listed above.

```
> C,pts,subst := EllipticGeneralFibreDeg2K3(S,B,[Ls[4],Lsd[4],
>   Ls[5],Lsd[5]]);
> C; Ambient(C);
Curve over Univariate rational function field over Rational Field defined by
(t^2 + 2*t + 1)/t^4*a^4 + (-2*t^3 - 2*t^2 - 6*t - 2)/t^4*a^3*c + (t^4 + 8*t^2 +
    4*t + 3)/t^4*a^2*c^2 + (-2*t^3 - 2*t^2 - 6*t - 2)/t^4*a*c^3 + (t^2 + 2*t +
    1)/t^4*c^4 - b^2
Projective Space of dimension 2 over Univariate rational function field over
Rational Field
Variables: t
Variables: a, b, c
The grading is:
    1, 2, 1
> pts;
[ (t : (-t^2 + 1)/t^2 : 1), (t : (t^2 - 1)/t^2 : 1), (1 : (-t - 1)/t^2 : 0),
(1 : (t + 1)/t^2 : 0) ]
> subst;
[
    $.1,
    (1/t*$.1^2 - $.1*$.3)/($.1 - 1/t*$.3),
```

(continues on next page)

(continued from previous page)

```
    $.3,
    ($.1^3*$.2 - $.1*$.2*$.3^2)/($.1^2 - 2/t*$.1*$.3 + 1/t^2*$.3^2)
]
```

C is given as a $y^2 =$ quartic curve in $\mathbf{P}(1,2,1)$ over $\mathbf{Q}(t)$. We transform to a Weierstrass form using $pts[4]$ as a base point

```
> E,Emp := EllipticCurve(C,pts[4]);
> E;
Elliptic Curve defined by y^2 + (-2*t^7 - 2*t^6 - 6*t^5 - 2*t^4)/(t^4 + 6*t^3 +
    4*t^2 + 2*t + 1)*x*y + (-4*t^19 - 20*t^18 - 52*t^17 - 92*t^16 - 108*t^15 -
    76*t^14 - 28*t^13 - 4*t^12)/(t^12 + 18*t^11 + 120*t^10 + 366*t^9 + 555*t^8 +
    588*t^7 + 496*t^6 + 324*t^5 + 171*t^4 + 74*t^3 + 24*t^2 + 6*t + 1)*y = x^3 +
2*t^8/(t^4 + 6*t^3 + 4*t^2 + 2*t + 1)*x^2 + (-4*t^24 - 32*t^23 - 112*t^22 -
    224*t^21 - 280*t^20 - 224*t^19 - 112*t^18 - 32*t^17 - 4*t^16)/(t^16 +
    24*t^15 + 232*t^14 + 1160*t^13 + 3268*t^12 + 5640*t^11 + 7096*t^10 +
    7128*t^9 + 5830*t^8 + 3976*t^7 + 2296*t^6 + 1112*t^5 + 452*t^4 + 152*t^3 +
    40*t^2 + 8*t + 1)*x + (-8*t^32 - 64*t^31 - 224*t^30 - 448*t^29 - 560*t^28 -
    448*t^27 - 224*t^26 - 64*t^25 - 8*t^24)/(t^20 + 30*t^19 + 380*t^18 +
    2650*t^17 + 11205*t^16 + 30376*t^15 + 56560*t^14 + 79960*t^13 + 91530*t^12 +
    87300*t^11 + 70824*t^10 + 49580*t^9 + 30090*t^8 + 15880*t^7 + 7280*t^6 +
    2872*t^5 + 965*t^4 + 270*t^3 + 60*t^2 + 10*t + 1) over Univariate rational
function field over Rational Field
```

We can now find minimising transformations to greatly simplify this Weierstrass equation but we will finish here.

126.8 Bibliography

- [BHPdV04] Barth, Hulek, Peters, and Van de Ven. *Compact Complex Surfaces*. Ergebnisse der Mathematik und ihrer Grenzgebiete 4. Springer, second edition, 2004. ISBN 3-540-00832-2.
- [Bec07] Tobias Beck. Formal Desingularization of Surfaces – The Jung Method Revisited –. Technical Report 2007-31, RICAM, December 2007. http://www.ricam.oeaw.ac.at/publications/reports/.
- [Bec08] Tobias Beck. Software Documentation. Technical Report 2008-8, RICAM, May 2008. http://www.ricam.oeaw.ac.at/publications/reports/.
- [BS08] Tobias Beck and Josef Schicho. Adjoint Computation for Hypersurfaces Using Formal Desingularizations. Technical Report 2008-2, RICAM, January 2008. http://www.ricam.oeaw.ac.at/publications/reports/.
- [BFHN] M. Besier, D. Festi, M.C. Harrison, and B. Naskrecki. Arithmetic and geometry of a K3 surface emerging from virtual corrections to Drell-Yan scattering. preprint arXiv:1908.01079 [math.AG].
- [dGP] W.A. de Graaf and J. Pilnikova. Parametrizing Del Pezzo surfaces of degree 8 using Lie algebras. http://arxiv.org/abs/math.NT/0512477.
- [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.
- [DES93] Decker, Ein, and Schreyer. Construction of surfaces in P4. *J. Algebraic Geometry*, 2:185–237, 1993.
- [Eis05] David Eisenbud. *The geometry of syzygies: a second course in commutative algebra and algebraic geometry*. Volume 225 of Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2005.

- [Els] Andreas-Stephan Elsenhans. Good models for cubic surfaces. (To appear). URL: http://www.staff.uni-bayreuth.de/ btm216/.
- [EJ10a] Andreas-Stephan Elsenhans and Jörg Jahnel. Cubic surfaces with a Galois invariant double-six. *Cent. Eur. J. Math.*, 8(4):646–661, 2010.
- [Fes] Dino Festi. A practical algorithm to compute the geometric picard lattices of K3 surfaces of degree 2. preprint arXiv:1808.00351 [math.AG].
- [GSHPBS12] J. Gonzalez-Sanchez, M. Harrison, I. Polo-Blanco, J. Schicho. "Algorithms For Del Pezzo Surfaces of Degree 5 (Construction, Parametrization)." JSC, 147(3), 342–353. 2012.
- [HS06] M.C. Harrison and J. Schicho. Rational Parametrisation for Degree 6 Del Pezzo Surfaces using Lie Algebras. In *Proceedings ISSAC'06*. 2006.
- [Har77] Robin Hartshorne. *Algebraic Geometry, GTM 52*. Springer, ASpringer, 1977.
- [Hor76] E. Horikawa. Algebraic Surfaces of General Type with Small c_1^2. II. *Invent. Math.*, 37:121–155, 1976.
- [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. URL: http://dx.doi.org/10.1515/crll.2000.064, doi:10.1515/crll.2000.064.
- [SD74] Bernard Saint-Donat. Projective Models of K-3 Surfaces. *Am. Jour. Math.*, 96(4):602–639, 1974.
- [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.
- [Sch00a] Josef Schicho. Proper parametrization of surfaces with a rational pencil. In *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation (St. Andrews)*, 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.

Chapter 127

Riemann Surfaces

127.1 Introduction to Riemann Surfaces

A Riemann surface object is defined by an affine plane equation $f(x, y) = 0$ in the following way. The Riemann surface X associated to the equation $f(x, y) = 0$, denoted $X : f = 0$ is the set of complex points of the non-singular model of the projective closure C of the affine curve defined by $f(x, y) = 0$, i.e., $X = C(\mathbf{C})$.

There are two different types of Riemann surface in Magma for which two different sets of algorithms are used, depending on the defining equation $f = 0$.

(i) The general Riemann surface type is defined by a geometrically irreducible function $f \in \mathbf{K}[x, y]$ over a number field K and an embedding $\sigma : K \to \mathbf{C}$;

(ii) The superelliptic Riemann surface type defined by an affine equation $y^m = p(x)$ where $p \in \mathbf{K}[x]$ is separable, $\mathbf{K} \subseteq \mathbf{C}$ and $m > 1$.

Note that, in practice, increasing the precision heavily impacts the running time of the algorithms. The algorithms for a general Riemann surface work reasonably well for up to a precision of several hundred digits, say up to $D \sim 500$, while the superelliptic case is considerably more efficient so that working to a precision of several thousand digits is possible, say up to $D \sim 2000$. Higher precisions are possible, but the computation may take considerable time.

The main application of this Riemann surface package is to numerically approximate a period matrix Ω that describes the analytic Jacobian $J(\mathbf{C}) = \mathbf{C}^g/\Omega\mathbf{Z}^{2g}$ of the curve C and associated Abel–Jacobi map $A : X \to J$ up to some prescribed precision D, where the number of decimal digits D can be specified by the user. The aim is that numerical computations should be correct up to an absolute error of 10^{-D+1} using heuristic error estimates.

The Riemann surface type is called `RieSrf`.

The algorithms for general Riemann surfaces are described in [Neu18, Chapter 4]. The algorithms for the superelliptic case are based on a paper by Molin and Neurohr [MN17]; more details can be found in [Neu18, Chapter 5]. Finally, the algorithms used for numerical integration are described in more detail in [Neu18, Chapter 3].

Example: Verbose Output for Riemann Surfaces `(ex-5b318d)`

For those who are interested in seeing what is happening behind the scenes, there is a verbose flag called `RieSrf` which has verbose levels in the range [0..3].

127.2 Creation Functions

127.2.1 Riemann Surfaces over Number Fields

As mentioned above, the very general class of Riemann surfaces over number fields can be defined by a geometrically irreducible polynomial $f \in \mathbf{Q}[x, y]$ or $f \in \mathbf{K}[x, y]$, together with the homomorphism

$\sigma : \mathbf{K} \to \mathbf{C}$, where K is a number field and σ is an infinite place for $\mathbf{K}$. The Riemann surface is constructed by analytic continuation of the multi-valued function $f(x,\ y(x)) = 0$ along the fundamental group of the extended complex plane punctured by the discriminant points of f with respect to the y.

The creation of a Riemann surface will automatically produce an associated big period matrix by numerically integrating a basis of holomorphic differentials along a homolgy basis using a numerical integration method `IntMethod`. More details on the bases that are being used is given below.

Recommended integration methods that are available in Magma are Gauss-Legendre quadrature ("GL"), Clenshaw–Curtis quadrature ("CC"), double-exponential integration ("DE") and mixed integration ("Mixed") which uses a combination of Gauss-Legendre and double-exponential integration. The default integration method is "Mixed" and this will outperform the other options in most cases. However, depending on the defining polynomial and the precision, there are cases where other choices may be better.

`RiemannSurface(f): RngMPolElt -> RieSrf`

```
Precision: RngIntElt                    Default: 30
IntMethod: MonStgElt                    Default: "Mixed"
```

Create a Riemann surface object defined by $X : f = 0$ where $f \in \mathbf{Q}[x, y]$ is geometrically irreducible.

`RiemannSurface(f, sigma): RngMPolElt, PlcNumElt -> RieSrf`

```
Precision: RngIntElt                    Default: 30
IntMethod: MonStgElt                    Default: "Mixed"
```

Create a Riemann surface object defined by $X : f = 0$ using the embedding $\sigma : K \to \mathbf{C}$, where K is a number field and $f \in \mathbf{K}[x, y]$ is geometrically irreducible.

Example: Riemann Surface over the Rationals (ex-c4a814)

We define the Riemann surface associated to the affine equation

$$f(x, y) = x^3 + x^2 + xy^3 - xy^2 + y^2 - y = 0$$

defined over $\mathbf{Q}$. Here the integration method is chosen at random to show the available options. The default is "Mixed" and is usually a good choice.

```
> Qxy<x,y> := PolynomialRing(Rationals(),2);
> f := x^3 + x^2 + x*y^3 - x*y^2 + y^2 - y;
> IntMeth := Random(["Mixed", "GL", "CC", "DE"]);
> X := RiemannSurface(f : Precision:=100, IntMethod:=IntMeth);
> X;

Riemann surface of genus 3 defined by: 0 = x^3 + x^2 + xy^3 - xy^2 + y^2 - y
and prescribed precision 100

Computed data:
 Complex fields: [ 100, 106, 126, 378 ]
 Discriminant points: true
 BasePoint: (-2.000000000, -0.8260051791)
 Path pieces: true
 Index path lists: true
 Closed chains: true
 Branch points: true
 Local monodromy: true
 Monodromy group: true
 Homology basis: true
 Holomorphic differential: true
```

(continues on next page)

(continued from previous page)

```
Integration method: CC
BigPeriod matrix: true
SmallPeriod matrix: false
Reduction matrix (real) false
Reduction matrices (complex) false
SheetToSheetIntegrals: true
```

Example: Riemann Surface over a Number Field (ex-6d46eb)

Taking an equation $f(x, y) = 0$ defined over a number field K together with a complex embedding σ of K is a more general way to define a Riemann surface. The Riemann surface defined by $f(x, y) = -s^2x^7 + 2sx^3y + y^3$, where s is a primitive element of the number field defined by $t^2 + 1$ is constructed in Magma as follows:

```
> Qt<t> := PolynomialRing(Rationals());
> K<s> := NumberField(t^2 + 1);
> Kxy<x,y> := PolynomialRing(K,2);
> f := -s^2*x^7 + 2*s*x^3*y + y^3;
> sigma := InfinitePlaces(K)[1];
> sigma;
1st place at infinity
> X := RiemannSurface(f,sigma : Precision:=100);
> X;

Riemann surface of genus 2 defined by: 0 = x^7 + 2*s*x^3*y + y^3
and prescribed precision 100

Computed data:
 Complex fields: [ 100, 106, 126, 378 ]
 Discriminant points: true
 BasePoint: (-2.000000000, -3.424546085 - 4.898716927*I)
 Path pieces: true
 Index path lists: true
 Closed chains: true
 Branch points: true
 Local monodromy: true
 Monodromy group: true
 Homology basis: true
 Holomorphic differential: true
 Integration method: Mixed
 BigPeriod matrix: true
 SmallPeriod matrix: false
 Reduction matrix (real) false
 Reduction matrices (complex) false
 SheetToSheetIntegrals: true
```

127.2.2 Superelliptic Riemann Surfaces

Another type of Riemann surface, for which a special set of faster algorithms is available, are *superelliptic Riemann surfaces*. These are defined by an affine equation of the form $y^m = p(x)$ where $m > 1$ and $p \in \mathbf{K}[x]$ is a separable polynomial (i.e., it has no multiple roots) of degree $\deg(p) \geq 3$ with $\mathbf{K} \subseteq \mathbf{C}$. Due to the special form of its defining equation and the resulting geometric properties, period matrices and the Abel-Jacobi map can be computed considerably faster, for higher genus, with more precision and more numerical stability.

Just as in the general case, a big period matrix will be computed using numerical integration at the time that a superelliptic Riemann surface is constructed. Available options for integration methods are

Gauss-Jacobi integration (J), double-exponential integration (DE) and a mixture of both (Mixed). For most examples, choosing mixed integration is the preferable option, as the algorithm will use Gauss-Jacobi for well-behaved integrands and switch to double-exponential for the difficult integrands.

RiemannSurface(p, m): RngUPolElt, RngIntElt -> RieSrf

```
Precision: RngIntElt                    Default: 30
IntMethod: MonStgElt                    Default: "Mixed"
```

Create a superelliptic Riemann surface object defined by $X : y^m = p(x)$.

RiemannSurface(L, m): SeqEnum[FldComElt], RngIntElt -> RieSrf

```
IntMethod: MonStgElt                    Default: "Mixed"
```

Create a superelliptic Riemann surface object defined by $X : y^m = p(x)$, where $p = L[n] \cdot \prod_{i=1}^{n-1}(x - L[i])$ with $n = \#L$. Here the sequence L contains the branch points of $p(x)$ together with its leading coefficient.

Example: Superelliptic Riemann Surface of Genus 12 (ex-d6ff62)

A superelliptic Riemann surface of genus 12 in precision 100 will be constructed using a random integration method:

```
> Q<x> := PolynomialRing(Rationals());
> p := x^5 - 5/2*x^4 + 5/3*x^3 - 1/6*x;
> m := 7;
> IntMeth := Random(["Mixed","GJ","DE"]);
> X1 := RiemannSurface(p,m : IntMethod := IntMeth, Precision:=100);
> Genus(X1);
12
> X1;

Superelliptic Riemann surface of genus 12 defined as degree 7 covering defined
by x^5 - 5/2*x^4 + 5/3*x^3 - 1/6*x
and prescribed precision 100


Computed data:
 Complex fields: [ 100, 110, 120 ]
 Discriminant points: true
 BasePoint: (-0.2637626158, 0.0000000000)
 Path pieces: false
 Index path lists: false
 Closed chains: false
 Branch points: true
 Local monodromy: false
 Monodromy group: false
 Homology basis: true
 Holomorphic differential: true
 Integration method: Mixed
 BigPeriod matrix: true
 SmallPeriod matrix: false
 Reduction matrix (real) false
 Reduction matrices (complex) false
 SheetToSheetIntegrals: false
```

For the next example the polynomial $p(x)$ above is taken over the complex field. Note that, in the output printing, the coefficients of the polynomial $p(x)$ are printed to only 10 decimal places.

```
> C<I> := ComplexField(100);
> X2 := RiemannSurface(ChangeRing(p,C),m);

Superelliptic Riemann surface of genus 12 defined as degree 7 covering defined
by $.1^5 - 2.5000000000*$.1^4 + 1.6666666667*$.1^3 -  0.1666666667$.1^1
and prescribed precision 100

Computed data:
 Complex fields: [ 100, 110, 120 ]
 Discriminant points: true
 BasePoint: (-0.2637626158, 0.0000000000)
 Path pieces: false
 Index path lists: false
 Closed chains: false
 Branch points: true
 Local monodromy: false
 Monodromy group: false
 Homology basis: true
 Holomorphic differential: true
 Integration method: Mixed
 BigPeriod matrix: true
 SmallPeriod matrix: false
 Reduction matrix (real) false
 Reduction matrices (complex) false
 SheetToSheetIntegrals: false
```

In the final example, a superelliptic Riemann surface is constructed given the branch points of $p(x)$ and its leading coefficient.

```
> pRts := Roots(ChangeRing(p,C));
> L := [ r[1] : r in pRts ] cat [ 1 ];
> X3 := RiemannSurface( L, m );
> X3;
Superelliptic Riemann surface of genus 12 defined as degree 7 covering defined
by x^5 - 2.5000000000*x^4 + 1.6666666667*x^3 - 0.1666666667*x
and prescribed precision 100

Computed data:
 Complex fields: [ 100, 110, 120 ]
 Discriminant points: true
 BasePoint: (-0.2637626158, 0.0000000000)
 Path pieces: false
 Index path lists: false
 Closed chains: false
 Branch points: true
 Local monodromy: false
 Monodromy group: false
 Homology basis: true
 Holomorphic differential: true
 Integration method: Mixed
 BigPeriod matrix: true
 SmallPeriod matrix: false
 Reduction matrix (real) false
 Reduction matrices (complex) false
 SheetToSheetIntegrals: false
```

127.3 Properties of Riemann Surfaces

127.3.1 Basic Invariants

In this section functions that access information about a Riemann surface are described. While most of these functions apply to both types of Riemann surface, some only apply to non-superelliptic Riemann surfaces.

BasePoint(X): RieSrf -> RieSrfPt

The base point of the Riemann surface X.

Genus(X): RieSrf -> RngIntElt

The genus of the Riemann surface X.

Degree(X): RieSrf -> RngIntElt

The degree of the holomorphic map to the projective line that is used to construct the Riemann surface X.

Precision(X): RieSrf -> RngIntElt

The precision to which the Riemann surface X has been constructed.

Embedding(X): RieSrf -> PlcNumElt

The complex embedding σ that is used to define the Riemann surface X.

BigPeriodMatrix(X): RieSrf -> Mtrx

The big period matrix associated to the Riemann surface X.

SmallPeriodMatrix(X): RieSrf -> Mtrx

Check: BoolElt	Default: true

The small period matrix associated with the Riemann surface X. If parameter `Check` is set to `true`, the small period matrix is checked to be a symmetric matrix (up to the prescribed precision) with a positive definite imaginary part.

FunctionField(X): RieSrf -> FldFun

The function field of the defining polynomial of the Riemann surface X. This is only available for non-superelliptic Riemann surfaces.

Example: Invariants of a Riemann Surface (ex-1117c8)

Some of these intrinsics will be demonstrated. Note that while the computation of the Riemann surface invariants has used a precision of 100 decimal digits, for reasons of brevity the period matrices are printed to just 12 decimal digits.

```
> Qt<t> := PolynomialRing(Rationals());
> K<s> := NumberField(t^2 + 1);
> y<x,y> := PolynomialRing(K,2);
> f := -s^2*x^7 + 2*s*x^3*y + y^3;
> sigma := InfinitePlaces(K)[1];
> X := RiemannSurface(f,sigma : Precision:=100);
>
> BasePoint(X);

(-2.000000000, -3.424546085 - 4.898716927*I)
>
> Genus(X);
2
>
> BigPeriodMatrix(X);
```

(continues on next page)

(continued from previous page)

```
[ 1.57333990636 - 1.14309835298*I, -2.91713261396 + 0.706473634625*I,
-1.34379270760 + 1.84957198760*I, -0.371415169614 - 1.14309835298*I ]
[ -1.15879743459 - 0.376516110443*I, -1.87497363525 - 1.82764759203*I,
-0.716176200654 - 2.20416370247*I, -1.15879743459 - 1.59494783846*I ]

>
>  SmallPeriodMatrix(X);
[ -0.164199030444 + 0.817677768561*I, 0.683580096956 + 0.0368649791982*I ]
[ 0.683580096956 + 0.0368649791982*I, 0.449259709133 + 0.252676326408*I
>
```

127.3.2 Fundamental Group

Intrinsics for constructing the branch points and fundamental group are described. While most of these functions apply to both types of Riemann surface, some only apply to non-superelliptic Riemann surfaces.

DiscriminantPoints(f): RngMPolElt -> SeqEnum[FldComElt]

DiscriminantPoints(f, sigma): RngMPolElt, PlcNumElt -> SeqEnum[FldComElt]

DiscriminantPoints(X): RieSrf -> SeqEnum[FldComElt]

The discriminant points of the defining polynomial with respect to the second variable embedded into $\mathbf{C}$ using an infinite place.

BranchPoints(X): RieSrf -> Tup

The projections of the ramification points of the Riemann surface X to the projective line.

RamificationPoints(X): RieSrf -> SeqEnum[RieSrfPt]

The ramification points of the ramified holomorphic map to the projective line used for the construction of X.

SingularPoints(X): RieSrf -> SeqEnum

While the Riemann surface itself is non-singular, this function returns the projective coordinates of the singular points of the projective closure of the underlying affine curve.

FundamentalGroup(P): SeqEnum[FldComElt] -> FldComElt, SeqEnum[FldComElt], SeqEnum[CPath], SeqEnum[SeqEnum[RngIntElt]]

```
BasePoint: MonStgElt                    Default: "Clever"
```

Given a set P of discriminant points this intrinsic returns a description of the fundamental group $\pi_1(\mathbf{C} \setminus P,\ x_0)$ of the punctured complex plane with base point x_0. The paths are constructed by modifying an euclidean spanning tree between the discriminant points by adding arcs and circles around the points. Options for **BasePoint** are **"Clever"** (positions the base point in the middle of the longest line segment), **"Left"** (base point on the negative real axis, left of all discriminant points) or a complex number specified by the user.

FundamentalGroup(X): RieSrf -> SeqEnum[CChain]

```
Extended: BoolElt                       Default: true
```

Given a Riemann surface X with discriminant points $D = \{x_1 \dots x_k\}$ and punctured surface S $= \mathbf{C} \setminus D$ return the chains of paths in the complex plane that generate the fundamental group $\pi_1(S, x_0)$, where x_0 is the x-coordinate of the base point of X. If the parameter **Extended** is set to **true**, a chain around infinity is returned as well, therefore generating $\pi_1(\mathbf{P}^1(\mathbf{C}) \setminus (S \cup \{\infty\}), x_0)$. The lifts of these chains represent a homology basis for X. This intrinsic is only available for non-superelliptic Riemann surfaces.

`MonodromyRepresentation(X): RieSrf -> SeqEnum`

A description of the monodromy of the Riemann surface X is returned.

Example: Fundamental Group of a Riemann Surface (ex-447d32)

```
> Qxy<x,y> := PolynomialRing(Rationals(),2);
> f := -x^7 + 2*x^3*y + y^3;
> X := RiemannSurface(f);
> FG := FundamentalGroup(X);
> FG;
[
    Closed chain consisting of 4 paths starting at -2.000000000 around
    -1.034563716
    Permutation: (2, 3)
    Integrals: true
    ,
    Closed chain consisting of 11 paths starting at -2.000000000 around
    0.0000000000
    Permutation: (1, 3)
    Integrals: true
    ,
    Closed chain consisting of 17 paths starting at -2.000000000 around
    -0.3196977700 - 0.9839285636*I
    Permutation: (2, 3)
    Integrals: true
    ,
    Closed chain consisting of 15 paths starting at -2.000000000 around
    0.8369796280 - 0.6081012948*I
    Permutation: (1, 2)
    Integrals: true
    ,
    Closed chain consisting of 13 paths starting at -2.000000000 around
    0.8369796280 + 0.6081012948*I
    Permutation: (2, 3)
    Integrals: true
    ,
    Closed chain consisting of 11 paths starting at -2.000000000 around
    -0.3196977700 + 0.9839285636*I
    Permutation: (1, 2)
    Integrals: true
    ,
    Closed chain consisting of 71 paths starting at -2.000000000 around Infinity
    Permutation: (1, 3, 2)
    Integrals: true

]
> d := #FG;
> &*[ FG[k] : k in [1..d-1] ]^(-1) eq FG[d];
true
> [ FG[k]`Center : k in [1..d-1] ] eq DiscriminantPoints(X);
true

> M := MonodromyRepresentation(X);
> M;
[
    (2, 3),
    (1, 3),
    (2, 3),
```

(continues on next page)

(continued from previous page)

```
    (1, 2),
    (2, 3),
    (1, 2),
    (1, 3, 2)
]

> #M eq #[ Chain`Permutation : Chain in FG | Order(Chain`Permutation) ne 1 ];
true
> &and[ Chain`StartPt eq Representation(BasePoint(X))[1] : Chain in FG ];
true
```

127.3.3 Basis for Period Matrix

`HolomorphicDifferentials(X): RieSrf -> Tup`

This intrinsic computes the basis of holomorphic differentials used for the computation of the period matrix and the Abel–Jacobi map. The actual information returned depends upon whether X is superelliptic or not. The general (non-superelliptic) case will be considered first. If the genus g of X equals the number of integral points on the interior of the Newton polygon associated with its defining polynomial, the basis used consists of $\omega = x^i y^j dx/d_y f(x,y)$ where $(i,j) \in \mathbf{Z}^2$ consists of aforementioned integral points. In this case a tuple is returned that contains a list of the pairs $[i,j]$ as first entry and $d_y f(x,y)$ as second entry.

In the case where the genus is not equal to the number of integral interior points if, for example, the projective curve is singular outside $\{(0:0:1),(0:1:0),(1:0:0)\}$, the basis used is that computed by the function field machinery, and embedded into $\mathbf{C}$ using the mapping σ. Let FFX denote `FunctionField(X)`. Specifically, this intrinsic returns the basis calculated by the intrinsic `BasisOfDifferentialsFirstKind(FFX)` represented as products of irreducible factors: a list of irreducible factors $[a_1,\ldots,a_k]$ for some $k > 0$ and a $(k \times g)$ integer matrix of multiplicities $(b_{i,j})$ such that a basis is given by $\omega_j = \prod_{i=1}^{k} a_i^{b_{i,j}},\ j = 1,..,g$.

Example: Holomorphic Differentials of a Plane Curve (ex-86f032)

In the following example the projective closure of the affine curve is singular at (0:0:1) and (0:1:0), so that the first basis described above can be used.

```
> Qxy<x,y> := PolynomialRing(Rationals(),2);
> f := -x^7 + 2*x^3*y + y^3;
> X := RiemannSurface(f);
> HolomorphicDifferentials(X);
<[ [ 2, 2 ], [ 4, 1 ] ], 2*x^3 + 3*y^2>
```

However, if the singularity is moved to $(-1:0:1)$, the basis as provided by the function field machinery is used. In the following example it can be easily seen that this function returns the corresponding product representation.

```
> f_trans := Evaluate(f,[x+1,y]);
> X_trans := RiemannSurface(f_trans);
> HolomorphicDifferentials(X_trans);
 < [ y, 2/3*x^3 + 2*x^2 + 2*x + y^2 + 2/3, x + 1 ],
    [ 1  0]
    [-1 -1]
    [ 1  3] >
> BasisOfDifferentialsFirstKind(FunctionField(X));
[ ((x) * (y) * (2/3*x^3 + y^2)^-1) d(x), ((x)^3 * (2/3*x^3 + y^2)^-1) d(x) ]
```

In the case of a superelliptic Riemann surface, a basis of the form $\omega_{i,j} = x^i dx/y^j$ is used. The condition on the pair $(i,j) \in \mathbf{Z}^2$ is provided by [Neu18, Proposition 5.18]. Hence in this case, a tuple containing all the pairs $[i,j]$ corresponding to such differentials is returned.

Example: Holomorphic Differentials of a Superelliptic Surface (ex-784cd3)

The holomorphic differentials of the superelliptic Riemann surface of genus 21 defined by $y^8 = B(8)$ where $B(n)$ is the n-th Bernoulli polynomial will be computed.

```
> p<x> := BernoulliPolynomial(8);
> p;
x^8 - 4*x^7 + 14/3*x^6 - 7/3*x^4 + 2/3*x^2 - 1/30
> X := RiemannSurface(BernoulliPolynomial(8),8);
> Genus(X);
21
> HolomorphicDifferentials(X);
<[ 0, 2 ], [ 0, 3 ], [ 1, 3 ], [ 0, 4 ], [ 1, 4 ], [ 2, 4 ], [ 0, 5 ],
 [ 1, 5 ], [ 2, 5 ], [ 3, 5 ], [ 0, 6 ], [ 1, 6 ], [ 2, 6 ], [ 3, 6 ],
 [ 4, 6 ], [ 0, 7 ], [ 1, 7 ], [ 2, 7 ], [ 3, 7 ], [ 4, 7 ], [ 5, 7 ]>
> Genus(X) eq #HolomorphicDifferentials(X);
true
```

`HomologyBasis(L): SeqEnum[GrpPermElt] -> SeqEnum[SeqEnum[RngIntElt]], Mtrx, Mtrx`

Check: BoolElt	Default: true

Given a sequence L containing permutation generators of the monodromy group, the homology basis used for computing the period matrix is returned. Two objects are returned: a sequence of integer sequences each of which defines an element of the homology group and an intersection matrix. If the parameter `Check` is set `true`, several tests for correctness are performed.

`HomologyBasis(X): RieSrf -> SeqEnum[SeqEnum[RngIntElt]], Mtrx, Mtrx`

Check: BoolElt	Default: true

Given a Riemann surface X the homology basis used for computing the period matrix is returned. Three objects are returned: a sequence of integer sequences each of which defines an element of the homology group, an intersection matrix, and the symplectic base change matrix that has been used for the computation of the big period matrix. If the parameter`Check` is set `true` several tests for correctness are performed.

Example: Homology Basis from Monodromy (ex-701b67)

In this example the homology basis will be computed from the monodromy representation.

```
> Qxy<x,y> := PolynomialRing(Rationals(),2);
> f := x^4*y^8 - x - y^8 + y^2 + 1;
> Chains := MonodromyRepresentation(f);
> MonRep := [ Ch`Permutation : Ch in Chains ];
> MonRep;
[
    (1, 2, 6, 8, 7, 3),
    (3, 5)(4, 6),
    (2, 4)(5, 7),
    (1, 2, 6, 8, 7, 3),
    (2, 4)(5, 7),
    (1, 2, 6, 8, 7, 3)(4, 5),
    (2, 5)(4, 7),
    (1, 5)(4, 8),
    (3, 5)(4, 6),
    (1, 2, 6, 8, 7, 3),
    (3, 5)(4, 6),
    (1, 7, 4, 3, 8, 2, 5, 6)
]
```

Using the permutation generators the monodromy group will be constructed.

```
> MonGroup := PermutationGroup< Degree(f,2) | MonRep >;
> IsTransitive(MonGroup);
true
> #MonGroup;
384

> Cycles, IntersectionMatrix := HomologyBasis(MonRep);
> Cycles[20];
[ 1, 4, 8, 1, 1 ]
```

The sequence of integers `Cycles`[20] above defines a non-trivial element of the homology group in the following way. It encodes a path starting on sheet 1, circling discriminant point $d4$ until one arrives at sheet 8, then continues circling discriminant point $d1$ until it returns to sheet 1. More explicitly, it can be expressed as the analytic continuation of the chain Chains[4]^3 * Chains[1]^3, starting and ending on the first sheet. Indeed, the permutation associated with the concatenation of chains turns out to be the identity:

```
> Chains[4]^3 * Chains[1]^3;
Closed chain consisting of 36 paths starting at
                              -0.5034246870 - 0.4967763264*I
Permutation: Id($)
```

In the case of a Riemann surface, the function `HomologyBasis` returns a third value, namely the symplectic base change matrix that has been used for the computation of the big period matrix. The Riemann surface package always uses the canonical basis of the form

$$J = \begin{pmatrix} 0 & I_g \\ -I_g & 0 \end{pmatrix}.$$

Example: Homology Basis for the Big Period Matrix (ex-8af6bb)

The homology basis for the big period matrix will be computed for the Riemann surface X defined by the polynomial $y^2 + x^3 - 1$.

```
> Qxy<x,y> := PolynomialRing(Rationals(),2);
> X := RiemannSurface(y^2 + x^3 - 1);

> Cycles, IntersectionMat, SymplecticTransform := HomologyBasis(X);
```

The canonical homology basis is now found by matrix multiplication.

```
> SymplecticTransform * IntersectionMat * Transpose(SymplecticTransform);

[ 0  1  0]
[-1  0  0]
[ 0  0  0]
```

127.4 Points on Riemann Surfaces

A point on a Riemann surface belongs to type `RieSrfPt`. At this point it is necessary to describe the representation of such points

(i) In the superelliptic case where $X : y^m = p(x)$ and $\deg(p) = n$, every finite point is uniquely represented by a pair $(x, y) \in \mathbf{C}^2$ of complex numbers. Points at infinity are represented by an index, an integer $s \in \{1, ..\delta = \gcd(m, n)\}$, in accordance with [Neu18, §5.4.3].

(ii) For general Riemann surfaces the situation is a little different. Let $X : f(x, y) = 0$ with $m = \deg(f, y)$ and denote by $\phi : X \to \mathbf{P}^1$ the corresponding holomorphic ramified map to the projective line defined by x. Algorithmically X is constructed via analytic continuation of $f(x, y(x)) = 0$ around the discriminant points of f with respect to y. This construction relies on the choice of a base point and an ordering of the sheets (the set of preimages under ϕ, indexed by $I_m = \{1 \dots m\}$). Now, every point on X is represented by a 2-tuple $\langle x,\ s\rangle$ where $x \in \mathbf{P}^1 = \mathbf{C} \cup \{\infty\}$ and $s \in I_m$.

127.4.1 Points

A number of ways of creating points on a Riemann surface will be described. Some functions returning points have already been met: BasePoint(X), RamificationPoints(X) and InfinitePoints(X). The most important way of creating a point on a Riemann surface is to coerce a sequence/tuple into the surface.

`IsCoercible(X, S): RieSrf, Any -> BoolElt, .`

Attempt to create a point on the Riemann surface X from the sequence or tuple S. In order to succeed S has to be a sequence or tuple of length 2 (resp. 3) that is coercible into a complex field and satisfies the affine (resp. projective) defining equation for X. A special case is the points at infinity of a superelliptic Riemann surfaces which may be defined by $[k]$ where k is an integer. If successful, the intrinsic returns `true` and the point on X. Otherwise `false` is returned.

`Point(X, S): RieSrf, SeqEnum -> RieSrfPt`

Attempt to create a point on the Riemann surface X from the sequence S by calling `IsCoercible(X,S)`. In order to succeed, S has to be a sequence of length 2 (resp. 3) that is coercible into a complex field and satisfies the affine (resp. projective) defining equation for X. A special case is that of the points at infinity of superelliptic Riemann surfaces which may be defined by $[k]$ where k is an integer.

`Point(X, S): RieSrf, Tup -> RieSrfPt`

If X is a non-superelliptic Riemann surface and S is a tuple defining a possible point of X, the intrinsic `IsCoercible(X,S` is called to test if S defines a valid point and if so, produce the point of X corresponding to S. The tuple S has to be of the form $< x,\ s >$ where x is either coercible into a complex field or `Infinity()` and $s \in \{1, .., m\}$. If successful the intrinsic returns `true` and the point on X. Otherwise `false` is returned.

Example: Creating a Point on a Riemann Surface (ex-6a04d2)

A point on the Riemann surface defined by $f = -x^7 + 2x^3y + y^3$ will be created.

```
> Qxy<x,y> := PolynomialRing(Rationals(),2);
> f := -x^7 + 2*x^3*y + y^3;
> X := RiemannSurface(f);
> X;
Riemann surface of genus 2 defined by: 0 = -x^7 + 2*x^3*y + y^3
and prescribed precision 30
```

The point of X defined by $x = [-2.000000000, -6.085518827]$ and $s = 2$ will be defined.

```
> b, pt := IsPoint(X, < [-2.000000000, -6.085518827], 2 >);
> b;
true
> pt;
(-2.000000000 - 6.085518827*I, -23.68417228 + 74.84801836*I)
```

127.4.2 Access Functions

`RiemannSurface(P): RieSrfPt -> RieSrf`

Given a point P belonging to the Riemann surface X, return X.

Representation(P): RieSrfPt -> Tup

Given a point P belonging to a Riemann surface X a representation of P is returned. If P can be uniquely represented by a pair $(x, y) \in \mathbf{C}^2$, a sequence containing these numbers is returned. In other cases, such as for infinite points or singular points of the underlying affine (projective) curve, the representation is a 2-tuple $\langle x, S\rangle$ where $x \in \mathbf{C} \cup \{\infty\}$ and $S \subset \{1, .., m\}$ is an indexed subset. The only special case is for points at infinity for superelliptic Riemann surfaces, which are represented by $\langle \infty, k\rangle$ where k in an integer.

Coordinates(P): RieSrfPt -> SeqEnum[FldComElt]

Given a point P belonging to a Riemann surface X, the coordinates of P in the projective closure of the affine curve are returned. Note that these coordinates are not necessarily unique, as they may correspond to singular points.

RamificationIndex(P): RieSrfPt -> RngIntElt

The ramification index of the point P belonging to a Riemann surface is returned.

PointsOverDiscriminantPoint(X, k): RieSrf, RngIntElt -> SeqEnum[RieSrfPt]

Given a Riemann surface X and an integer k in the range $[0 \dots t]$, where t is the number of discriminant points for X, the points lying over the k-th discriminant point are returned. If k is zero, the points lying over infinity are returned.

RandomPoint(X): RieSrf -> RieSrfPt

```
Ht    : RngIntElt                    Default: 10^5
Finite: BoolElt                      Default: true
```

A randomly generated point on the Riemann surface X is returned. If the parameter `Finite` is set to `false` a random infinite point is returned. The second parameter `Ht`, allows the user to bound the absolute value of the x-coordinate.

Example: Point Access Functions on a Riemann Surface (ex-cb6770)

The various intrinsics dealing with points will be demonstrated.

```
> Qxy<x,y> := PolynomialRing(Rationals(),2);
> f := -4*x^4 - 5*x^3*y + x^3 + 2*x^2*y^2 - 5*x^2*y + 3*x^2
>        + 3*x*y^3 + x*y - 5*x - 8*y^3 - 3;
> X := RiemannSurface(f);
```

An easy way of defining a point on a Riemann surface X is via a pair of coordinates, using the fiber attribute of X.

```
> L := X`Fiber(0);
> Pts := [ X![0,L[i]]: i in [1..#L] ];
> Pts;
[
    (0.0000000000, -0.7211247851),
    (0.0000000000, 0.3605623926 - 0.6245123832*I),
    (0.0000000000, 0.3605623926 + 0.6245123832*I)
]
```

Using the Riemann surface `Print` command it is easy to control the precision used when printing objects. The various ways of printing the infinite points will also be demonstrated.

```
> Print(Pts[1] : Precision := Precision(X));
(0.00000000000000000000000000000, -0.72112478515370419116081915539)
> Representation(Pts[1]);
[ 0.00000000000000000000000000000000000000000000000000000000000,
-0.72112478515370419116081915539005479419593462674967528 8 ]
```

Next the various ways of printing the infinite points are shown.

```
> InfinitePoints(X);
[
    Point at infinity on sheets {@ 1 @},
    Point at infinity on sheets {@ 2, 3 @},
    Y-infinite point over x = 2.666666667 on sheets {@ 2 @}
]
> < Representation(P): P in InfinitePoints(X) >;
<<Infinity, {@ 1 @}>, <Infinity, {@ 2, 3 @}>,
<2.666666666666666666666666666666666666666666666666666667, {@ 2 @}>>

> X!<Infinity(),1> eq InfinitePoints(X)[1];
true
> X!<Infinity(),2> eq X!<Infinity(),3>;
true
> X!<8/3,2> eq InfinitePoints(X)[3];
true
```

Using the intrinsic `Coordinates`, the projective coordinates of the points can be retrieved.

```
> [ Coordinates(Pt): Pt in InfinitePoints(X) ];
[
    [ 0.75000000000000000000000000000000, 1.00000000000000000000000000000000,
    0.00000000000000000000000000000000 ],
    [ -1.00000000000000000000000000000000, 1.00000000000000000000000000000000,
    0.00000000000000000000000000000000 ],
    [ 0.00000000000000000000000000000000, 1.00000000000000000000000000000000,
    0.00000000000000000000000000000000 ]
]
> X![3/4,1,0] eq InfinitePoints(X)[1];
true
> X![-1,1,0] eq InfinitePoints(X)[2];
true
> X![0,1,0] eq InfinitePoints(X)[3];
true
```

Next the ramification points will be printed.

```
> RamificationPoints(X);
[
    (-0.9945334268 - 0.01315512874*I, 0.05451089484 - 0.1643916971*I),
    (-0.9945334268 + 0.01315512874*I, 0.05451089484 + 0.1643916971*I),
    (-0.5252377944 + 0.03223401255*I, 0.01450803309 - 0.2043107259*I),
    (-0.5252377944 - 0.03223401255*I, 0.01450803309 + 0.2043107259*I),
    (0.8365927809 + 0.5016034840*I, -0.4936743728 + 0.5748084711*I),
    (0.8365927809 - 0.5016034840*I, -0.4936743728 - 0.5748084711*I),
    (0.6518240243 - 0.8906693804*I, -1.581225731 - 0.3052027833*I),
    (2.549953730, 19.14988442),
    (0.6518240243 + 0.8906693804*I, -1.581225731 + 0.3052027833*I),
    Point at infinity on sheets {@ 2, 3 @}
]

The discriminant points and the points lying over them will be examined.

> DiscriminantPoints(X)[1];
-0.994533426780725243594285430344169336513181415875424160 -
0.013155128735979994941172544525719487958174959849345786 2*I
```

(continues on next page)

(continued from previous page)

```
> PointsOverDiscriminantPoint(X,1);
[
    (-0.9945334268 - 0.01315512874*I, 0.05451089484 - 0.1643916971*I),
    (-0.9945334268 - 0.01315512874*I, 0.07106577813 + 0.3329009545*I)
]
```

Points lying over Infinity() are obtained as follows:

```
> PointsOverDiscriminantPoint(X,0);
[
    Point at infinity on sheets {@ 1 @},
    Point at infinity on sheets {@ 2, 3 @}
]
```

Random finite or infinite points on X can be obtained as follows:

```
> P1 := RandomPoint(X: Finite);
> P1;
(-2.285969242 + 16.06877365*I, -1.575257283 + 21.29854014*I)


> P2 := RandomPoint(X: Finite := false);
> P2;
Y-infinite point over x = 2.666666667 on sheets {@ 2 @}
```

127.5 Divisors on Riemann Surfaces

A rather simple concept of a divisor for a Riemann surface is supported. Here the divisors are realised as finite sums of points with integer coefficients. The type name for these Riemann surface divisors is `DivRieSrfElt`. The main application of these divisors is to the computation of the Abel–Jacobi map.

`Divisor(S, V): SeqEnum[RieSrfPt], SeqEnum[RngIntElt] -> DivRieSrfElt`

Given a sequence S of points P_i belonging to Riemann surface X and a sequence V of integers n_i, construct the formal divisor $\sum_i n_i P_i$.

`ZeroDivisor(X): RieSrfElt -> DivRieSrfElt`

Construct the zero divisor for the Riemann surface X.

`RiemannSurface(D): DivRieSrfElt -> RieSrf`

The Riemann surface associated with the divisor D is returned.

`Support(D): DivRieSrfElt -> SeqEnum[RieSrfPt], SeqEnum[RngIntElt]`

Given a divisor $D = \sum_i n_i P_i$ with all n_i nonzero, the sequence of points P_i and the sequence of their multiplicities n_i are returned.

`Degree(D): DivRieSrfElt -> RngIntElt`

The sum of the multiplicities n_i of the points P_i supporting the divisor $D = \sum_I N_I p_I$ is returned.

`RandomDivisor(X, d): RieSrf, RngIntElt -> RieSrfDivElt`

```
Ht  : RngIntElt                 Default: 10^5
Zero: BoolElt                   Default: true
```

Given a Riemann surface X and a positive integer d return a random divisor for X of degree d. The maximum size of the coefficients can be bounded by assigning a positive integer to the parameter `Ht`. If parameter `Zero` is set to `true`, the degree of the returned divisor will be zero.

127.6 Abel–Jacobi Map

This section covers the construction of the Abel–Jacobi map for Riemann surfaces (intrinsic `AbelJacobi`). For a Riemann surface X and a base point $P_0 \in X$ the Abel–Jacobi map is defined as

$$A : X \to J = \mathbf{C}^g/\Lambda, \quad P \mapsto (\int_{P_0}^{P} \omega_j)_{j=1,..,g} \pmod{\Lambda}$$

The Abel–Jacobi map extends to divisors on X by linearity and is independent of the base point for divisors of degree 0. In particular, $A(D) = 0$ if and only if D is a principal divisor.

`AbelJacobi(P): RieSrfPt -> Mtrx`

`AbelJacobi(DX): DivRieSrfElt -> Mtrx`

`AbelJacobi(DF, X): DivFunElt, RieSrf -> Mtrx`

```
Reduction: MonStgElt                    Default: "Complex"
Method   : MonStgElt                    Default: "Swap"
```

Let X be a Riemann surface, let P be a point on X and let B be the base point returned by `BasePoint`. These intrinsics construct the Abel–Jacobi maps with respect to base point B given either a point P, a divisor DX of X, or a divisor DF of an associated function field of X, respectively. The map is returned as a matrix.

In the case of non-superelliptic Riemann surfaces two options are provided as to how the Abel–Jacobi map of ramification points should be handled. If the parameter `Method` is assigned the value "Swap" and $P = (x, y)$ then $A(P)$ is obtained by computing $A'(y, x)$ where A' is the Abel–Jacobi map on $X' : f(y, x) = 0$; compare [Neu18, §4.9.3]. This method is numerically robust and is the default option. However, if $\deg(f, x) \gg \deg(f, y)$, it might be a bad choice. The other option is "Direct" which will try adaptive double-exponential integration as described in [Neu18, §4.9.5]; it is completely heuristic and somewhat unstable but often returns the correct results.

We provide several options for reduction modulo the period lattice. These are selected by assigning one of the following three values to the parameter `Reduction`.

(i) "None" : This will simply compute the vector of integrals in $\mathbf{C}^{g\times 1}$ and not reduce it at all. Note that this option is used internally and the results may not be mathematically correct.

(ii) "Real" : Returns the image of the Abel–Jacobi map in the canonical torus $\mathbf{R}^{2g}/\mathbf{Z}^{2g}$ as described in [Neu18, §4.9.6].

(iii) "Complex" : The default value for `Reduction`. Returns the image of the Abel–Jacobi map in $\mathbf{C}^g/\Lambda$ where $\Lambda = \langle 1, \tau \rangle$ and τ is the small period matrix associated to X.

`AbelJacobi(P, Q): RieSrfPt, RieSrfPt -> Mtrx`

```
Reduction: MonStgElt                    Default: "Complex"
Method   : MonStgElt                    Default: "Swap"
```

Given points P and Q belonging to a Riemann surface X, the Abel–Jacobi map for the divisor $[Q - P]$ on X is computed. The possible values for the parameters `Reduction` and `Method` are as given in the description of the previous Abel–Jacobi intrinsic.

`AbelJacobi(D, P): DivRieSrfElt, RieSrfPt -> Mtrx`

```
Reduction: MonStgElt                    Default: "Complex"
Method   : MonStgElt                    Default: "Swap"
```

Let X be a Riemann surface with base point P and divisor D. The Abel–Jacobi map is constructed on X with respect to the base point P. The possible values for the parameters `Reduction` and `Method` are as given in the description of the previous Abel–Jacobi intrinsic.

Example: Abel-Jacobi Map on a Superelliptic Curve (ex-909191)

Using the Abel–Jacobi map for superelliptic curves of the form $y^m = p(x)$ we can easily verify that the divisors are of the form $D = P_k - P_j$ for m-torsion divisors for ramification points P_k and P_j. We do this for the following genus 16 example with precision set to 100 almost instantaneously.

```
> Qx<x> := PolynomialRing(Rationals());
> p := &*[ x - k : k in [1..10] ];
> m := 4;
> X := RiemannSurface(p,m : Precision:=100);
> D := X![Random([1..5]),0] - X![Random([6..10]),0];
> A := m * AbelJacobi(D:Reduction:="Real");
> &and[ Abs(c-Round(c)) lt X`Error : c in Eltseq(A) ];
true
```

or even more directly

```
> A := AbelJacobi(m * D : Reduction := "Complex");
> &and[ Abs(c) lt X`Error : c in Eltseq(A) ];
true
```

Or that the divisor $D = \sum_{k=1}^{\delta} P_\infty^{(k)}$ is a m/δ-torsion divisor, where $\delta = \gcd(m, \deg(p))$:

```
> D := Divisor( InfinitePoints(X), [1, 1] );
> E := X![1] + X![2];
> D eq E;
true
> AJ := AbelJacobi(D : Reduction:="Real");
```

The projective coordinates of the points at infinity may be obtained by using the intrinsic `Coordinates`. However, in our example these coordinates define a singular point on the projective curve and so cannot be used to define a point on the Riemann surface:

```
> X!Coordinates(X![1]);
Runtime error in '!': Singular point of the projective closure.
> Coordinates(Random(InfinitePoints(X))) in SingularPoints(X);
true
```

Whenever projective coordinates correspond one-to-one to points on the Riemann surface, i.e., they are non-singular points of the projective curve, they can be used to obtain the coordinates of the points at infinity on the Riemann surface:

```
> X := RiemannSurface(LegendrePolynomial(4),4);
> InfinitePoints(X);
[
    1st point at infinity,
    2nd point at infinity,
    3rd point at infinity,
    4th point at infinity
]
> [ X!Coordinates(P) : P in InfinitePoints(X) ] eq InfinitePoints(X);
true
```

Example: Abel-Jacobi Map of a Principal Divisor (ex-860b42)

Let us now consider how to use the Abel–Jacobi map for general Riemann surfaces defined over number fields. For example, we may verify that the Abel–Jacobi map of the principal divisor $(y - y_1)/(y - y_2)$ lies in the period lattice.

```
> Qxy<x, y> := PolynomialRing(Rationals(), 2);
> f := 2*x^3*y^5 + x^2*y^6 - 1;
> X := RiemannSurface(f);
> m := X`Degree[1];
> n := X`Degree[2];
> f := ComplexPolynomial(X);
> C<I> := BaseRing(Parent(f));
```

We now generate two random complex numbers y1 and y2.

```
> Ht := 10^5;
> y1 := Random([-Ht..Ht])/Random([1..Ht]) + I*Random([-Ht..Ht])/Random([1..Ht]);
> y2 := Random([-Ht..Ht])/Random([1..Ht]) + I*Random([-Ht..Ht])/Random([1..Ht]);
> F1 := RootsNonExact(UnivariatePolynomial(Evaluate(f, [Parent(f).1, y1])));
> F2 := RootsNonExact(UnivariatePolynomial(Evaluate(f, [Parent(f).1, y2])));
> S1 := []; S2 := [];
> for j in [1..n] do
>   Append(~S1, X![F1[j], y1]);
>   Append(~S2, X![F2[j], y2]);
> end for;
> D := &+S1 - &+S2;
> A := AbelJacobi(D);
> &and[ Abs(c) lt X`Error : c in Eltseq(A) ];
true
```

Example: Abel-Jacobi Map of Function Field Divisors (ex-3a3dc0)

One of the highlights of the package is that one can directly compute the Abel–Jacobi map of function field divisors. We will illustrate this using a genus 4 example defined over a degree 3 number field:

```
> Qt<t> := PolynomialRing(Rationals());
> K<s> := NumberField(t^3 + 43274/61569*t^2 + 1024/6841*t + 9122/6841);
> Kxy<x, y> := PolynomialRing(K,2);
> f := 1/26116*(9985*s^2 + 49265*s - 17877)*x^3*y^4 + 1/34512*(26093*s^2 -
>    6604*s + 46087)*x*y^4 + 1/23026*(22829*s^2 + 2142*s - 18264)*x*y +
>    1/46947*(5011*s^2 + 5883*s - 39463)*x + 1/85753*(2423*s^2 - 21329*s
>    + 41637);
> sigma := InfinitePlaces(K)[1];
> X := RiemannSurface(f, sigma : Precision := 50);
> FF<v, w> := FunctionField(X);
/* Take all infinite places in the function field */
> D1 := &+SetToSequence(Set(Poles(v) cat Poles(w)));
/* Take all infinite points on the Riemann surface */
> D2 := &+InfinitePoints(X);
> A1 := AbelJacobi(D1, X);
> A2 := AbelJacobi(D2);
> &and[ Abs(c) lt X`Error : c in Eltseq(A1-A2) ];
true
```

Another good test is to vereify that divisors of meromorphic functions always map to the period lattice under the Abel–Jacobi map.

```
> A1 := AbelJacobi(Divisor(v), X);
> &and[ Abs(c) lt X`Error : c in Eltseq(A1) ];
true
> A2 := AbelJacobi(Divisor(w), X);
> &and[ Abs(c) lt X`Error : c in Eltseq(A2) ];
```

(continues on next page)

(continued from previous page)

```
true
> A3 := AbelJacobi(Divisor(s^2*v^2*w+v/w/3), X);
> &and[ Abs(c) lt X`Error : c in Eltseq(A3) ];
true
```

One also might want to compute the Abel–Jacobi map of the canonical divisor, reduce it modulo the small period matrix and then compute its multi-dimensional Theta function.

```
> D := CanonicalDivisor(FF);
> A := AbelJacobi(D, X : Reduction := "Complex");
```

Finally we define the desired characteristic and compute the Theta function!

```
> char := Transpose(Matrix([[ 1/2 : j in [1..2*Genus(X)]]]));
> Theta(char, A, SmallPeriodMatrix(X));
0.29810003713725582196195932655751593711064828563915 +
0.68480591255255827524715345957080419035255625041761*I
```

127.7 Period Matrix Functions

Since the period matrices that are computed for Riemann surfaces describe the analytic Jacobian of the associated algebraic curve, several functions from Section Analytic Jacobians of Hyperelliptic Curves "Analytic Jacobians of Hyperelliptic Curves" can be applied as well. In particular, if τ_1, τ_2 denote small and Ω_1, Ω_2 denote big period matrices corresponding to genus g Riemann surfaces, the following intrinsics apply:

(i) `AnalyticHomomorphism` (τ_1, τ_2)

(ii) `IsIsomorphicSmallPeriodMatrices` (τ_1, τ_2)

(iii) `IsIsomorphicBigPeriodMatrices` (Ω_1, Ω_2)

(iv) `IsIsogenousPeriodMatrices` (Ω_1, Ω_2)

Moreover, the endomorphism ring of a big period matrix Ω can be computed by using the intrinsic

(v) `EndomorphismRing`(Ω)

Finally, the evaluation of the multidimensional theta function associated to a small period matrix τ with characteristic p (a $(2g \times 1)$-matrix) and at a z (a $(g \times 1)$-matrix) is performed by the intrinsic:

(vi) `Theta(p, z, \tau)`

Example: Period Matrices of a Hyperelliptic Curve (ex-7d1774)

For a hyperelliptic curve we may obtain period matrices in three different ways now:

```
> C<I> := ComplexField(100);
> Cx<x> := PolynomialRing(C);
> f := BernoulliPolynomial(10);
> A := AnalyticJacobian(Evaluate(f, x));
> X := RiemannSurface(f, 2 : Precision := 100);
> Qxy<x,y> := PolynomialRing(Rationals(), 2);
> Y := RiemannSurface(y^2 - Evaluate(f, x) : Precision := 100);
> IsIsomorphicSmallPeriodMatrices(SmallPeriodMatrix(A), SmallPeriodMatrix(Y));
true
[ 0  0  0  0  0  0  1 -1]
[ 0  1 -1 -1  0  1 -1  0]
[ 0 -1 -1 -1  0  0  0  0]
[-1 -1 -1 -1 -1  0  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  1  0  0  0  0]
[ 0  0  1  1  0  0  0  0]
[ 0  0  1  1  1 -1  0  0]
[ 1  1  1  1  0  0  0  0]
> IsIsomorphicSmallPeriodMatrices(SmallPeriodMatrix(X), SmallPeriodMatrix(Y));
true
[ 1 -1  0  0  0 -1  0  0]
[ 1  0  1  1  0  0  1  1]
[ 1  0  0  0  0  0  1 -1]
[ 0  0  1  0  1  1  0  0]
[-1  1  0  0  0  0  0  0]
[-1  0 -1 -1  0  0 -1  0]
[-1  0 -1  0  0  0 -1  0]
[-1  0  0  0  0  0  0  0]
```

127.8 Bibliography

- [MN17] Pascal Molin and Christian Neurohr. Computing period matrices and the Abel-Jacobi map of superelliptic curves. *Mathematics of Computation*, Dec 2017. URL: http://dx.doi.org/10.1090/mcom/3351, doi:10.1090/mcom/3351.
- [Neu18] Christian Neurohr. *Efficient integration on Riemann surfaces & applications.* Dissertation, Carl von Ossietzky Universität Oldenburg, 2018. URL: http://oops.uni-oldenburg.de/3607.

Chapter 128

Hilbert Series of Polarised Varieties

128.1 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.

128.1.1 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):

$$C\colon (y^2 = f) \subset \mathbb{P}(1, 1, g + 1)$$

where the coordinates on $\mathbb{P}(1, 1, g+1)$ 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 *Interpreting the Hilbert Numerator*, 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 $\mathbb{P}(1, 1, g + 1)$ 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 $k = \mathbb{C}$, 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 $X \subset \mathbb{P}^N$. 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 $X \subset \mathbb{P}^N(w_0, \ldots, w_N)$ 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 *Hilbert Series and Hilbert Polynomials*.

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 *Schemes* and then attempt to construct a Weierstrass model of the hyperelliptic curve as described in Chapter Hyperelliptic Curves 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).

128.1.2 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 *Hilbert Series and Graded Rings* 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 *Baskets of Singularities* 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 *Generic Polarised Varieties* 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 *Subcanonical Curves* 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 *K3 Surfaces*. 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 *The K3 Database* 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 *Fano 3-folds*, the latter in Section *Calabi–Yau 3-folds*. 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 *Building Databases* 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 [KS00b] notwithstanding.

128.2 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.)

128.2.1 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 $x^2, xy, \ldots, t^2$, 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 *Interpreting the Hilbert Numerator* 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

$$P(t) = 1 + t + 2t^2 + 4t^3 + 5t^4 + 7t^5 + 9t^6 + \ldots$$

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 *Subcanonical Curves*, 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): RngUPolElt, SeqEnum -> UserProgram`

HilbertFunction(Q, V): SeqEnum, SeqEnum -> UserProgram

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): RngUPolElt, SeqEnum -> FldFunRatUElt

HilbertSeries(Q, V): SeqEnum, SeqEnum -> FldFunRatUElt

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.

128.2.2 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

$$P(t) = \frac{1 - t^{d_1} - t^{d_2} - \cdots - t^{d_r} + t^e \pm \cdots \pm t^k}{(1 - t^{w_0}) \cdots (1 - t^{w_N})}.$$

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): RngUPolElt, SeqEnum -> RngUPolElt, SeqEnum

HilbertSeriesMultipliedByMinimalDenominator(Q, V): SeqEnum, SeqEnum -> RngUPolElt, SeqEnum

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): FldFunRatUElt, SeqEnum -> FldFunRatUElt

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: Hilbert Function of a Genus 4 Curve (ex-bc959f)

We reproduce the example described in the introduction to this Section *Hilbert Series and Graded Rings*.

```
> 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 *Hilbert Series and Graded Rings* 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): FldFunRatUElt -> SeqEnum

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: First Generators from a Hilbert Series (ex-12902f)

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): RngUPolElt -> RngIntElt

ApparentEquationDegrees(f): RngUPolElt -> RngIntElt

ApparentSyzygyDegrees(f): RngUPolElt -> RngIntElt

If $f = f(t)$ is a polynomial of the form

$$f = 1 - \sum_{i=1}^{N_0} a_{0,i}t^i + \sum_{k=N_0+1}^{N_1} a_{1,i}t^i - \cdots + (-1)^{k-1} \sum_{i=N_{k-2}+1}^{N_{k-1}} a_{k-1,i}t^i + (-1)^k t^{N_k}$$

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.

128.3 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.

128.3.1 Point Singularities

The point singularities that are allowed are always finite cyclic quotient singularities. In the surface case (over the complex numbers $\mathbf{C}$) 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 $a, b \in \{0, \ldots, r-1\}$. 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 $\frac{1}{r}(a_1, \ldots, a_k)$, 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 $p = \frac{1}{r}(a_1, \ldots, a_k)$, the *index of* p is the positive integer r and the *polarisation of* p is the sequence $[a_1, \ldots, a_k]$.

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 *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 $\{0, \ldots, r-1\}$. When we need to include the eigenspace of A in the singularity, we use the symbol $\frac{1}{r}(a_1, \ldots, a_k)_n$.

Example: Surface and 3-Fold Point Singularities (ex-6bdfc9)

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
```

Creation of Point Singularities

`Point(r, n, Q): RngIntElt, RngIntElt, SeqEnum -> GRPtS`

`Point(r, Q): RngIntElt, RngIntElt, SeqEnum -> GRPtS`

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 $n = -1$.

Accessing the Key Data and Testing Equality

`Dimension(p): GRPtS -> RngIntElt`

Dimension of the variety on which the singularity p lies.

`Index(p): GRPtS -> RngIntElt`

Index of the singularity p.

`Polarisation(p): GRPtS -> SeqEnum`

Polarisation sequence of the singularity p.

`Eigenspace(p): GRPtS -> RngIntElt`

Eigenspace of the polarising divisor of the point singularity p.

`p eq q: GRPtS, GRPtS -> BoolElt`

Return `true` if and only if the two point singularities p, q have the following attributes equal: dimension, index, polarisation, eigenspace.

Identifying Special Types of Point Singularity

`IsIsolated(p): GRPtS -> BoolElt`

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): GRPtS -> BoolElt`

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): GRPtS -> BoolElt`

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): GRPtS -> RngIntElt`

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): GRPtS -> SeqEnum`

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): GRPtS -> BoolElt`

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.

128.3.2 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: Curve Singularity with Transverse Type `(ex-117c52)`

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
```

Creation of Curve Singularities

`Curve(d, p, m): FldRatElt, GRPtS, FldRatElt -> GRCrvS`

`Curve(d, p, N): FldRatElt, GRPtS, RngIntElt -> GRCrvS`

`Curve(d, p, N, t): FldRatElt, GRPtS, RngIntElt, RngIntElt -> GRCrvS`

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 $m = N/t$.

Accessing the Key Data and Testing Equality

`Degree(C): GRCrvS -> RngIntElt`

The degree of the curve singularity C.

`TransverseType(C): GRCrvS -> GRPtS`

The transverse type of the curve singularity C, that is, a point surface singularity that is the general transverse section of C.

`TransverseIndex(C): GRCrvS -> RngIntElt`

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): GRCrvS -> RngIntElt`

The invariant N of the curve singularity C.

`Index(C): GRCrvS -> RngIntElt`

The invariant t (sometimes called τ) of the curve singularity C that is determined by the indices of dissident points on C.

`MagicNumber(C): GRCrvS -> RngIntElt`

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): GRCrvS -> RngIntElt`

The dimension of the curve singularity C (currently always 3).

`IsCanonical(C): GRCrvS -> BoolElt`

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: GRCrvS, GRCrvS -> BoolElt`

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).

128.3.3 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.

Creation and Modification of Baskets

`Basket(Q): SeqEnum -> GRBskt`

`Basket(Q1, Q2): SeqEnum, SeqEnum -> GRBskt`

The basket of singularities where Q, Q_1, Q_2 are sequences of point or curve singularities.

`EmptyBasket(): . -> GRBskt`

The basket of singularities containing no singularities.

`MakeBasket(Q): SeqEnum -> GRBskt`

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 $\frac{1}{Q[1]}(Q[2], \ldots, Q[N])$. Note that a local polarisation n cannot be included in this constructor: the default value $n = -1$ is always assumed.

`Points(B): GRBskt -> SeqEnum`

The sequence of point singularities of the basket B.

`Curves(B): GRBskt -> SeqEnum`

The sequence of curve singularities of the basket B.

Tests for Baskets

`IsIsolated(B): GRBskt -> BoolElt`

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): GRBskt -> BoolElt`

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): GRBskt -> BoolElt`

Return `true` if and only if all singularities of the basket B are terminal 3-fold singularities.

`IsCanonical(B): GRBskt -> BoolElt`

Return `true` if and only if all singularities of the basket B are canonical singularities.

128.3.4 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): GRCrvS -> SeqEnum`

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): GRCrvS -> SeqEnum`

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): GRCrvS -> SeqEnum`

A sequence of points, each of may appear on the curve singularity C as a dissident point.

`PossibleSimpleCanonicalDissidentPoints(C): GRCrvS -> SeqEnum`

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.

128.4 Generic Polarised Varieties

Recall from Section *Key Warning and Disclaimer* that, despite some ambiguity, we regard the following as being equivalent: polarised varieties X, A; schemes in wps $X \subset \mathbb{P}^N(w_0, \ldots, w_N)$ 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): RngIntElt, SeqEnum, RngUPolElt -> GRSch`

The polarised variety of dimension d, with weights given by the sequence W of positive integers and with Hilbert numerator the univariate polynomial n.

128.4.1 Accessing the Data

`Weights(X): GRSch -> SeqEnum`

The weights of the polarised variety X.

`Degree(X): GRSch -> FldRatElt`

The degree of the polarised variety X.

`Basket(X): GRSch -> Bskt`

The basket of singularities of the polarised variety X.

`RawBasket(X): GRSch -> SeqEnum`

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): GRSch -> RngIntElt`

The dimension of the polarised variety X.

`Codimension(X): GRSch -> RngIntElt`

The codimension of the polarised variety X.

`HilbertNumerator(X): GRSch -> RngUPolElt`

`Numerator(X): GRSch -> RngUPolElt`

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): GRSch -> SeqEnum`

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): GRSch -> RngUPolElt`

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): GRSch -> Tup`

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): GRSch -> FldFunRatUElt`

The Hilbert series of the polarised variety X expressed as a rational function.

`InitialCoefficients(X): GRSch -> SeqEnum`

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): GRSch -> RngIntElt`

`ApparentEquationDegrees(X): GRSch -> RngIntElt`

`ApparentSyzygyDegrees(X): GRSch -> RngIntElt`

`BettiNumbers(X): GRSch -> RngIntElt`

If $n(t)$ is the Hilbert numerator of X and is of the form

$$n = 1 - \sum_{i=1}^{N_0} a_{0,i}t^i + \sum_{k=N_0+1}^{N_1} a_{1,i}t^i - \cdots + (-1)^{k-1} \sum_{i=N_{k-2}+1}^{N_{k-1}} a_{k-1,i}t^i + (-1)^k t^{N_k}$$

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 $a_{k-1,i}$.

128.4.2 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: GRSch, GRSch -> BoolElt`

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): GRSch -> BoolElt`

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): GRSch -> SeqEnum`

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): GRSch, RngIntElt`

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): GRSch, RngIntElt`

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): GRSch`

Remove any weights from X whose presence is not required to keep the Hilbert numerator of X a polynomial.

128.5 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.

128.5.1 Creation of Subcanonical Curves

`SubcanonicalCurve(g, d, Q): RngIntElt, RngIntElt, SeqEnum -> GRCrvK`

The subcanonical curve C, D of genus g, degree d, and initial Hilbert series coefficients Q.

`IsSubcanonicalCurve(g, d, Q): RngIntElt, RngIntElt, SeqEnum -> BoolElt, GRCrvK`

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): RngIntElt, RngIntElt -> RngUPolElt`

The Hilbert polynomial $mt + 1 - g$ of a divisor of degree m on a curve of genus g.

`IsEffective(C): GRCrvK -> BoolElt`

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 $p_1 > 0$.

128.5.2 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): RngIntElt -> SeqEnum`

`EffectiveSubcanonicalCurves(g, d): RngIntElt, RngIntElt -> SeqEnum`

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): RngIntElt -> SeqEnum`

`IneffectiveSubcanonicalCurves(g, d): RngIntElt, RngIntElt -> SeqEnum`

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).

128.6 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 *Generic Polarised Varieties* 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.

128.6.1 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 *Baskets of Singularities*, 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): RngIntElt, GRBskt -> GRK3`

`K3Surface(g, B): RngIntElt, SeqEnum -> GRK3`

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): GRK3 -> GRK3`

A new K3 surface that carries exactly the same data as the K3 surface X.

128.6.2 Accessing the Key Data

`Genus(X): GRK3 -> RngIntElt`

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): GRK3 -> RngIntElt`

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): GRK3 -> RngIntElt`

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): GRK3 -> RngIntElt`

The number assigned to the K3 surface X in the low codimension lists of Altinok–Fletcher–Reid.

128.6.3 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): GRK3, RngIntElt -> GRK3`

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): GRK3, RngIntElt -> GRK3`

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.

128.7 Weil Polynomials

The characteristic polynomial of the Frobenius on the etale cohomology is called a Weil polynomial. This section contains some auxiliary routines for simpler handling of these polynomials. All routines make frequently use of `PowerSumToCoefficients`, `CoefficientsToElementarySymmetric` and `ElementarySymmetricToPowerSums`. They were used in [EJ10b] to study K3 surfaces.

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

Set the verbose printing level for the analysis of weil polynomials. Maximal value 2.

`HasAllRootsOnUnitCircle(f): RngUPolElt -> BoolElt`

Given a polynomial f with rational coefficients this routine checks that the complex roots of the polynomial are all of absolute value 1. The algorithm does not use floating point approximations.

`FrobeniusTracesToWeilPolynomials(tr, q, i, deg): SeqEnum, RngIntElt, RngIntElt, RngIntElt -> SeqEnum`

```
KnownFactor    : RngUPolElt                    Default: 1
WeilPolynomials: Verbose                       Default: 2
```

Given the sequence of Frobenius traces on the i-th etale cohomology this function returns a list of possible characteristic polynomials. Here q is the order of the basefield and deg is the degree of the characteristic polynomial. In the case of even i the sign in the functional equation has to be determined. This is done by checking the absolute values of the roots of the hypothetical polynomials. In the case that this does not work both candidates are returned. In the case of contradictory data an empty sequence is returned.

If a factor of the characteristic polynomial is known it can be given as optional argument.

`WeilPolynomialToRankBound(f, q): RngUPolElt, RngIntElt -> RngIntElt`

Given a polynomial f this routine counts the zeros (with multiplicity) of f that are q times a root of unity. This is an upper bound for the Picard rank of the corresponding algebraic surface; according to the Tate conjecture, it is precisely the Picard rank.

`ArtinTateFormula(f, q, h20): RngUPolElt, RngIntElt, RngIntElt -> RngIntElt, RngIntElt`

Given a Weil polynomial corresponding to H^2 of an algebraic surface with Hoge number $h^{2,0}$ this routine evaluates the Artin-Tate formula. If the Tate conjecture holds, the two values returned are the arithmetic Picard rank, and the absolute value of the discriminant of the Picard group times the order of the Brauer group of the surface.

`WeilPolynomialOverFieldExtension(f, deg): RngUPolElt, RngIntElt -> RngUPolElt`

Given the characteristic polynomial f of the Frobenius on an etale cohomology group. This routine returns the characteristic polynomial of the deg times iterated Frobenius.

`CheckWeilPolynomial(f, q, h20): RngUPolElt, RngIntElt, RngIntElt -> BoolElt`

```
SurfDeg: RngIntElt                    Default: -1
```

Given a polynomial f this routine checks several conditions that must be satisfied by the characteristic polynomial of the Frobenius on H^2 of an algebraic surface. Here q is the size of the basefield, $h20$ is the Hodge number $h^{2,0}$ and `SurfDeg` is the degree of the surface (-1 means degree unknown). For example, this routine can be used to determine the sign in the functional equation.

The routine checks the valuation of the roots at all places (including p and ∞). It also checks the functional equation and the Artin-Tate conditions (see [EJ10b] for details).

Example: Weil Polynomial of a Quartic Surface (ex-01b9a6)

```
> q1<t> := PolynomialRing(RationalField());
> z4<x,y,z,w> := PolynomialRing(IntegerRing(),4);
> f := x^4 + x^3*z + x^2*y^2 + x^2*y*w + x^2*z^2 + x^2*z*w +
>      x*y^3 + x*y*z*w + x*y*w^2 + x*z^2*w + y^3*w + y^2*z^2 +
>      y^2*z*w + y^2*w^2 + y*z^2*w + z^4 + z*w^3;
> Tr := [];
> for i := 1 to 10 do
>    P3 := ProjectiveSpace(GF(2^i),3);
>    S := Scheme(P3,f);
>    time
>    Tr[i] := #Points(S) - 1 - 2^(2*i);
> end for;
> cpl := FrobeniusTracesToWeilPolynomials(Tr, 2, 2, 22: KnownFactor := t-2);
> cpl;
[
    t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
         + 16384*t^7 - 32768*t^6 - 65536*t^5 + 262144*t^3
         + 524288*t^2 + 1048576*t - 4194304,
    t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
         - 16384*t^7 + 32768*t^6 + 65536*t^5 - 262144*t^3
         - 524288*t^2 - 1048576*t + 4194304
]
```

Here the sign in the functional equation can not be determined by the absolute values of roots test. Further point counting would be very slow, as the coefficients of t^{11}, t^{10}, t^9, t^8 are zero, thus point counting over $\mathbf{F}_{2^{15}}$ would be necessary to determine the sign in this way. Instead, we can try to use more invariants of the surface.

```
> cpl_2 := [wp : wp in cpl | CheckWeilPolynomial(wp,2,1: SurfDeg := 4)];
> cpl_2;
[
    t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
         + 16384*t^7 - 32768*t^6 - 65536*t^5 + 262144*t^3
         + 524288*t^2 + 1048576*t - 4194304
]
> WeilPolynomialToRankBound(cpl_2[1],2);
2
> ArtinTateFormula(cpl_2[1],2,1);
1 4
> ArtinTateFormula(WeilPolynomialOverFieldExtension(cpl_2[1],2),2^2,1);
2 68
```

Thus the minus sign in the functional equation is correct. The surface has arithmetic Picard rank 1 and relative to the Tate conjecture the geometric Picard group has rank 2 and discriminant either -17 or -68 (depending on the order of the Brauer group).

Example: Bounding Picard Rank with Fewer Point Counts (ex-43f336)

Advanced example: Bounding Picard rank with less point counting.

The computation is based on the same equation as the example above. Now we do not use the most costly Frobenius trace, but we assume additional cyclotomic factors.

```
> q1<t> := PolynomialRing(RationalField());
> Tr := [ 1, 5, 19, 33, 11, -55, 435, -31, -197]; /* First 9 Frob traces */
> cycl := [ Evaluate(CyclotomicPolynomial(i),t/2)*2^EulerPhi(i)
>         : i in [1..100] | EulerPhi(i) lt 23];
```

(continues on next page)

(continued from previous page)

```
> cycl[1] := cycl[1]^2;
> cycl[2] := cycl[2]^2; /* linear factors need multiplicity two */
> hyp := &cat [FrobeniusTracesToWeilPolynomials(Tr, 2, 2, 22 :
>                       KnownFactor := (t-2)*add) : add in cycl];
> hyp := SetToSequence(Set(hyp));
> hyp;
[
    t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
         - 16384*t^7 + 32768*t^6 + 65536*t^5 - 262144*t^3
         - 524288*t^2 - 1048576*t + 4194304
]
```

Using only the number of point over $\mathbb{F}_2, \ldots, \mathbb{F}_{2^9}$ we can prove that either the Picard rank is bounded by 2 or the correct Weil polynomial is in the list.

```
> CheckWeilPolynomial(hyp[1],2,1:SurfDeg := 4);
false
```

The invariants of the surface and the hypothetical Weil polynomial are contradictory. Thus the Picard rank is at most 2.

128.8 Point Counting on Degree Two K3 Surfaces

There are various methods to count points on varieties over finite fields. This section describes a p-adic method to count the number of points on K3 surfaces of degree 2 over finite fields. The result of the point count is returned as the characteristic polynomial of the Frobenius on the second cohomology vector space.

SetVerbose("Degree2K3", v): MonStgElt, RngIntElt

Set the verbose printing level for the p-adic point counting algorithm. Maximal value 2.

WeilPolynomialOfDegree2K3Surface(f6): RngMPolElt -> RngUPolElt, RngUPolElt

KnownFactor: RngUPolElt	Default: false

Given a homogeneous polynomial f_6 of degree 6 in 3 variables over a finite field, this function considers the double cover of the projective plane $w^2 = f_6$. If this surface is smooth or has only du Val singularities, it is a model of a K3 surface. In this case a p-adic point counting method is applied to the surface. Further, the Frobenius action on the singularities is analyzed. If the singularities are harder, then zero is returned twice. Otherwise, the characteristic polynomial of the Frobenius on the second cohomology of the surface is returned as two factors. The first factor corresponds to the potentially singular model given. If a divisor of the first factor is known the computation can be sped up by assigning it to KnownFactor.

The second factor corresponds to the exceptional divisors generated by blowing up all singularities. The singularity analysis is done without performing any blow-up steps. See section *Singularity Properties* for details. If the Frobenius action on the singularities is too complex to be covered by the current implementation then 0 is returned.

NonOrdinaryPrimes(f6, lim): RngMPolElt, RngIntElt -> SeqEnum

A prime number p is called non-ordinary for a K3 surface, if and only if the number of points on the reduction of the K3 surfaces modulo p is congruent to 1 modulo p. This intrinsic returns a list of all the odd non-ordinary primes of the surface $w^2 = f_6$ up to lim.

NumbersOfPointsOnDegree2K3Surface(f6, p, d): RngMPolElt, RngIntElt, RngIntElt -> SeqEnum

Returns as a sequence the numbers of points on the K3-surface $w^2 = f_6$ over $\mathbf{F}_p, \ldots, \mathbf{F}_{p^d}$.

Example: Weil Polynomial of a Degree 2 K3 Surface (ex-9195d9)

```
> r3<x,y,z> := PolynomialRing(Integers(),3);
> f6 := x*y*z*(x+y+z)*(x+2*y+5*z)*(3*x+2*y-7*z);
> wf1,wf2 := WeilPolynomialOfDegree2K3Surface(PolynomialRing(GF(61),3)!f6);
> wf1;
t^7 - 15*t^6 - 4575*t^5 - 234423*t^4 + 14299803*t^3 + 1038438075*t^2 +
    12668944515*t - 3142742836021
> wf2;
t^15 - 915*t^14 + 390705*t^13 - 103276355*t^12 + 18899572965*t^11 -
    2536322691903*t^10 + 257859473676805*t^9 - 20223550149795135*t^8 +
    1233636559137503235*t^7 - 58529201194634875705*t^6 +
    2142168763723636450803*t^5 - 59396497539609919772265*t^4 +
    1207728783305401702036055*t^3 - 17001105180376039344046005*t^2 +
    148152488000419771426686615*t - 602486784535040403801858901
> PolynomialToPowerSums(wf1);
[ 15, 9375, 912519, 2895543, 1009235775, 16650196047, 4570755285559 ]
Factorization(wf1);
[
    <t - 61, 1>,
    <t^6 + 46*t^5 - 1769*t^4 - 342332*t^3 - 6582449*t^2 + 636908686*t +
     51520374361, 1>
]
```

From the power sums we can derive the number of points on the singular model over $\mathbb{F}_{61}, \ldots, \mathbb{F}_{61^7}$ as $1+15+61^2$, $1+9375+61^4$,..., $1+4570755285559+61^{14}$. Further, the desingularization of the model will produce 15 exceptional lines. Finally, the factorization of `wf1` shows that the K3 surface and its reduction modulo 61 have geometric Picard rank 16.

```
> r3<x,y,z> := PolynomialRing(Integers(),3);
> NonOrdinaryPrimes(x^6+y^6+z^6 + 3*(x*y*z)^2,1000);
[ 3, 61, 89, 103, 419, 593, 613, 719 ]
> NonOrdinaryPrimes(x^6+y^6+z^6 - 3*(x*y*z)^2,1000);
[ 3, 7, 11, 19, 23, 31, 43, 47, 59, 67, 71, 79, 83, 103, 107, 127, 131, 139,
151, 163, 167, 179, 191, 199, 211, 223, 227, 239, 251, 263, 271, 283, 307,
311, 331, 347, 359, 367, 379, 383, 419, 431, 439, 443, 463, 467, 479, 487,
491, 499, 503, 523, 547, 563, 571, 587, 599, 607, 619, 631, 643, 647, 659,
683, 691, 719, 727, 739, 743, 751, 787, 811, 823, 827, 839, 859, 863, 883,
887, 907, 911, 919, 947, 967, 971, 983, 991 ]
```

The examples show that some K3 surfaces have very few non-ordinary primes and others have a lot of them.

128.9 The K3 Database

The K3 database in Magma is a collection of 24,099 K3 surfaces. Recall from Section *K3 Surfaces* the meaning of K3 surface in this context, and from Section *Key Warning and Disclaimer* 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 S := [ X : X in D | Genus(X) eq 3 ];
Time: 139.510
> time T := [K3Surface(D,[4],i) :i in [1..NumberOfK3Surfaces(D,[4])]];
Time: 0.500
```

128.9.1 Searching the K3 Database

In this section a simple example is presented of extracting a K3 surface with particular properties from the K3 database. 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); see Section *Key Warning and Disclaimer* for why a surface in the database cannot be assumed to exist.

Example: Accessing the K3 Database (ex-d3e9e7)

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 $[2, 3, 3, \dots]$.

```
> 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() -> DB`

The database of K3 surfaces.

`Number(D, X): DB, GRK3 -> RngIntElt, GRK3`

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): DB, GRK3 -> RngIntElt, GRK3`

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: Number and Index of a K3 Surface (ex-9475c3)

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
```

128.9.2 Working with the K3 Database

`K3Surface(D, i): DB, RngIntElt -> GRK3`

The ith K3 surface in the K3 database D.

`K3Surface(D, Q, i): DB, SeqEnum, RngIntElt -> GRK3`

The ith K3 surface in the K3 database D among those with index suite $Q = [g1 + 1, g2 + 1, ...]$.

`K3Surface(D, g, i): DB, RngIntElt, RngIntElt -> GRK3`

The ith K3 surface in the K3 database D among those with genus $g \geq -1$.

`K3Surface(D, g1, g2, i): DB, RngIntElt, RngIntElt, RngIntElt -> GRK3`

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): DB, SeqEnum -> GRK3`

The K3 surface in the K3 database D having the weights specified in the sequence W.

`K3Surface(D, g, B): DB, RngIntElt, GRBskt -> GRK3`

`K3Surface(D, g, B): DB, RngIntElt, SeqEnum -> GRK3`

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).

128.10 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 *Generic Polarised Varieties* 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: Fano 3-Folds with the Same Basket `(ex-109f4a)`

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 $f, a, r-a$.

128.10.1 Creation: $f = 1, 2$ or ≥ 3

`Fano(f, B, g): RngIntElt, GRBskt, RngIntElt -> GRFano`

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): RngIntElt, GRBskt -> GRFano`

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): GRFano -> RngIntElt`

The Fano index f of the Fano 3-fold X.

`FanoGenus(X): GRFano -> RngIntElt`

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): GRFano -> RngIntElt`

The smallest possible value for the Fano genus of the Fano 3-fold X.

`BogomolovNumber(X): GRFano -> FldRatElt`

The intersection number $A(c_1(X)^2 - 3c_2(X))$ for the polarised Fano 3-fold X, A.

`IsBogomolovUnstable(X): GRFano -> BoolElt`

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.

128.10.2 A Preliminary Fano Database

`FanoDatabase() -> DB`

The database of Fano 3-folds.

`Fano(D, i): DB, RngIntElt -> GRFano`

The ith Fano 3-fold in the Fano database D.

`Fano(D, f, i): DB, RngIntElt, RngIntElt -> GRFano`

The ith Fano 3-fold in the Fano database D that has Fano index f.

`Fano(D, f, Q, i): DB, SeqEnum, RngIntElt -> GRFano`

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).

128.11 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 *Generic Polarised Varieties* 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): RngIntElt, RngIntElt, GRBskt -> GRCY`

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): GRCY -> RngIntElt, RngIntElt`

```
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): RngIntElt, RngIntElt, GRBskt -> RngIntElt, RngIntElt`

```
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).

128.12 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 [KS00b] range.

128.12.1 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 *The K3 Database* 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.

Creating Many K3 Surfaces

`CreateK3Data(g): RngIntElt -> SeqEnum`

`CreateK3Data(g, r): RngIntElt, RngIntElt -> SeqEnum`

`CreateK3Data(g, B): RngIntElt, SeqEnum -> SeqEnum`

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.

K3 Surfaces as Records

`K3SurfaceToRecord(X): GRK3 -> Rec`

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): Rec -> GRK3`

The K3 surface with the same data as that of the record x in K3 record format.

Writing K3 Surfaces to a File

`WriteK3Data(Q, F): SeqEnum, MonStgElt`

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.

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…".

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): DB, RngIntElt -> Tup`

`K3SurfaceRaw(D, Q, i): DB, SeqEnum, RngIntElt -> Tup`

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): Tup -> GRK3`

The K3 surface with the same data as that of the tuple x that is in the raw K3 database format.

128.12.2 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.

128.13 Bibliography

- [Alt98] Selma Altnok. *Graded Rings Corresponding to Polarised K3 Surfaces and Q-Fano 3-folds*. PhD thesis, University of Warwick, 1998. http://www.maths.warwick.ac.uk/ miles/students/Selma/.
- [ABR02] Selma Altnok, 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.
- [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.

- [EJ10b] Andreas-Stephan Elsenhans and Jörg Jahnel. Weil polynomials of K3 surfaces. In *Algorithmic number theory*, volume 6197 of Lecture Notes in Computer Science, 126–141. Berlin, 2010. Springer.
- [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.
- [KS00b] 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*, 2003.
- [Rei00] Miles Reid. Graded rings and birational geometry. In K. Ohno, editor, *Proc. of algebraic geometry symposium (Kinosaki)*, pages 1 – 72. 2000.
- [Suz] K. Suzuki. On Fano indices of Q-Fano 3-folds. Available as math.AG/0210309 on arXiv.
- [SB] B. Szendroi and A. Buckley. Orbifold Riemann-Roch for threefolds with an application to Calabi-Yau geometry. math.AG/0309033.

Chapter 129

Toric Varieties

129.1 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.

129.1.1 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: Projective Plane from Its Fan `(ex-93014b)`

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;
```

(continues on next page)

(continued from previous page)

```
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,
```

(continues on next page)

(continued from previous page)

```
    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 ]
```

129.1.2 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 *Invariant Divisors and Riemann-Roch Spaces*.

Example: Danilov's Nonprojective Toric Variety (ex-e700da)

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
```

(continues on next page)

(continued from previous page)

```
> 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).

Example: Self-Intersection in a Hirzebruch Surface (ex-ee24e7)

We calculate the self-intersection of the zero section in the Hirzebruch surface $\mathbb{F}_2$, that is, the scroll $\mathbb{P}(O(-2) \oplus O)$ over $\mathbb{P}^1$. First we construct $X = \mathbb{F}_2$:

```
> X<x,y,u,v>:=AbsoluteRationalScroll(Rationals(),[2,0]);
> X;
Toric variety of dimension 2 over Rational Field
Variables: x, y, u, v
The components of the irrelevant ideal are:
    (y, x), (v, u)
The 2 gradings are:
    1,  1, -2,  0,
    0,  0,  1,  1
```

The zero locus is the third toric divisor.

```
> ray:=Cone(Rays(X)[3]);
> form:=IntersectionForm(X,ray);
> D:=Weil(Divisor(X,[0,0,1,0]));
> phi:=WeilToClassLatticesMap(X);
> form * phi(D);
-2
```

Example: Fano Index of a Toric Fano 3-Fold (ex-9b8afb)

We calculate the Fano index of a nonsingular toric Fano variety. We're interested in number 15 in the database of smooth toric Fano threefolds. First we fetch the corresponding Fano polytope P from the database:

```
> P:=PolytopeSmoothFanoDim3(15);
> P;
3-dimensional polytope P with 5 vertices:
    ( 1,  0,  0),
    ( 0,  1,  0),
    ( 0,  0,  1),
    (-1, -1,  1),
    ( 0,  0, -1)
```

We convert the polytope P into a toric variety X whose fan is generated by the cones spanning the faces of P:

```
> X<x,y,u,z,v>:=ToricVariety(Rationals(),SpanningFan(P));
> X;
Toric variety of dimension 3 over Rational Field
Variables: x, y, u, z, v
The components of the irrelevant ideal are:
    (v, u), (z, y, x)
The 2 gradings are:
    0, 0, 1, 0, 1,
    1, 1, 0, 1, 1
> IsNonsingular(X);
true
> IsFano(X);
true
```

The variety X has Fano index 2:

```
> K:=CanonicalDivisor(X);
> phi:=WeilToClassLatticesMap(X);
> fano_idx:=GCD(Eltseq(phi(Weil(-K))));
> fano_idx;
2
> half_K:=phi(Weil(-K)) / 2;
> A:=Divisor(X,half_K @@ phi);
> A;
Weil divisor A with coefficients:
    0, 0, 0, 1, 1
> IsCartier(A);
true
> IsAmple(A);
true
> bool:=AreLinearlyEquivalent(-K, 2 * A);
> bool;
true
```

Next we calculate the anti-canonical degree $(-K)^2$ and first few terms of the Hilbert series $\mathrm{Hilb}(X, -K)$:

```
> Degree(-K);
56
> HilbertCoefficients(-K,5);
[ 1, 31, 145, 399, 849, 1551 ]
```

We recognise X as 2-35 in the Mori–Mukai classification.

129.1.3 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: Cox's Quotient Construction (ex-bb7120)

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)
```

(continues on next page)

(continued from previous page)

```
> 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 $\mathbb{Z}$-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 $\mathbb{Z}$-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.)

129.2 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`.

129.2.1 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): [TorCon] -> TorFan`

```
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): [TorLatElt], [[RngIntElt]] -> TorFan`

```
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): TorCon -> TorFan`

The fan comprising all faces of the cone C.

`FanOfAffineSpace(n): RngIntElt -> TorFac`

The standard fan of affine space of dimension n, where n is a positive integer.

`FanOfWPS(W): SeqEnum -> TorFan`

A standard fan for the weighted projective space with weights W, a sequence of positive integers.

`FanOfProjectiveSpace(n): RngIntElt -> TorFac`

The standard fan of projective space of dimension n, a positive integer.

`FanOfFakeProjectiveSpace(W, Q): SeqEnum, SeqEnum -> TorFan`

A standard fan for the fake weighted projective space with weights W, a sequence of positive integers, and finite cyclic group actions given by Q, a sequence of sequences of rational numbers. The finite cyclic actions are determined as follows: for $\mathbb{Z}/r$ to act diagonally with weights $(a_1, \ldots, a_n)$, include the sequence `[a1/r,...,an/r]` as an element of Q.

`ZeroFan(L): TorLat -> TorFan`

The fan in the toric lattice L supported at the origin.

`NormalFan(F, C): TorFan, TorCon -> TorFan, Map`

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.

`NormalFan(P): TorPol -> TorFan`

`DualFan(P): TorPol -> TorFan`

The normal fan to the polyhedron P; in particular, the rays of the fan are normal to the facets of P. The resulting fan lies in the lattice dual to the ambient lattice of P.

`SpanningFan(P): TorPol -> TorFan`

The toric fan that spans the polyhedron P; in particular, the rays of the fan are generated by the vertices of P. The resulting fan lies in the same ambient lattice as P.

Example: Spanning Fan of a Polytope (ex-f5bea2)

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 $\mathbb{F}_1$ 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
```

`FanWithWeights(W): SeqEnum -> TorFan`

```
ample     : [RngIntElt]                    Default: the sum of weights
define_fan: BoolElt                        Default: false
```

The fan generated by the weights (or gradings) W, where W is a sequence of sequences of integers. The optional parameter `ample` can be used to specify the ample divisor to use. By default this is equal to the sum of the weights. If the optional parameter `define_fan` is set to `true` then the verification that the weights define a fan is skipped.

`Blowup(F, v): TorFan, TorLatElt -> TorFan`

The blowup of the toric fan F at the point v that is an element of the ambient toric lattice of F.

Example: Fan from Weight Data (ex-2a8220)

We construct a toric Fano three-fold X directly from the weight (or grading) data:

```
> F:=FanWithWeights([
>     [ 1, 1,-1, 0,-2, 0 ],
```

(continues on next page)

(continued from previous page)

```
>     [ 0, 0, 1, 1, 0, 0 ],
>     [ 0, 0, 0, 0, 1, 1 ]
> ]);
> X<[x]>:=ToricVariety(Rationals(),F);
> X;
Toric variety of dimension 3 over Rational Field
Variables: x[1], x[2], x[3], x[4], x[5], x[6]
The components of the irrelevant ideal are:
    (x[4], x[3]), (x[5], x[2]), (x[3], x[2]), (x[6], x[5], x[1]),
    (x[6], x[4], x[1])
The 3 gradings are:
    0, 0, 1, 1, 0, 0,
    0, 1, 0, 0, 1, 0,
    1, 0, 0, 2, 1, 1
> IsFano(X);
true
> IsNonsingular(X);
false
> IsQFactorial(X);
true
> IsTerminal(X);
true
```

Now we begin by taking $P = \mathbb{P}(1,1,1,2)$ and blowing up twice. We obtain a $\mathbb{Q}$-factorial terminal Fano three-fold Y.

```
> P:=WPS(Rationals(),[1,1,1,2]);
> F:=Fan(P);
> F;
Fan F with 4 rays:
    ( 0, -1, -1),
    (-1,  0,  1),
    (-1,  1,  0),
    ( 1,  0,  0)
and 4 cones with indices:
    [ 1, 2, 3 ],
    [ 1, 2, 4 ],
    [ 1, 3, 4 ],
    [ 2, 3, 4 ]
> N:=Ambient(F);
> pt1:=N ! [1,-1,0];
> pt2:=N ! [-1,0,0];
> Y1:=Blowup(P,pt1);
> Y<[y]>:=Blowup(Y1,pt2);
> Y;
Toric variety of dimension 3 over Rational Field
Variables: y[1], y[2], y[3], y[4], y[5], y[6]
The components of the irrelevant ideal are:
    (y[6], y[5]), (y[6], y[4]), (y[5], y[3]), (y[4], y[2], y[1]),
    (y[3], y[2], y[1])
The 3 gradings are:
    0, 0, 0, 1, 0, 1,
    0, 0, 1, 0, 1, 0,
    1, 1, 1, 2, 0, 0
> IsFano(Y);
true
```

We want to show that X and Y are isomorphic. Since they are both Fano, we will work with the polytopes generated by the rays of the fans. It is then trivial to construct an isomorphism between these two polytopes; equivalently, we have an isomorphism between the fans of X and Y.

```
> PX:=Polytope(Rays(Fan(X)));
> PY:=Polytope(Rays(Fan(Y)));
> bool,phi:=IsIsomorphic(PX,PY);
> bool;
true
> phi;
Mapping from: 3-dimensional toric lattice Z^3 to TorLat: N given by a rule
> DefiningMatrix(phi);
[ 0 -1  0]
[-1  0  1]
[ 1 -1  0]
> Image(phi,Fan(X)) eq Fan(Y);
true
```

IsInSupport(v, F): TorLatElt, TorFan -> BoolElt, RngIntElt

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.

OneSkeleton(F): TorFan -> TorFan

The fan generated by the rays of the fan F.

Fan(F1, F2): TorFan, TorFan -> TorFan

F1 * F2: TorFan, TorFan -> TorFan

Fan(Q): [TorFan] -> TorFan

F ^ n: TorFan, RngIntElt -> TorFan

The nth Cartesian product of the fan F.

F eq G: TorFan, TorFan -> BoolElt

Return **true** if and only if the two fans F and G are equal as objects in Magma (not simply isomorphic as fans).

129.2.2 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 $\mathbb{C}^*$ 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 $\mathbb{C}^*$). 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 $\mathbb{C}^*$ 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): TorFan, RngIntElt -> TorFan

The fan generated by cones of the fan F of dimension at most the integer n.

C in F: TorCon, TorFan -> BoolElt

Return **true** if and only if the cone C is one of the cones of the fan F.

Cones(F): TorFan -> SeqEnum

A sequence of the maximal cones of the fan F.

Cones(F, i): TorFan, RngIntElt -> SeqEnum

The sequence of all i-dimensional cones in the fan F.

`ConesOfCodimension(F, i): TorFan, RngIntElt -> SeqEnum`

The sequence of all codimension i cones in the fan F.

`MaxCones(F): TorFan -> SeqEnum`

The cones of maximum dimension in the fan F.

`AllCones(F): TorFan -> SeqEnum`

A sequence of all the cones of the fan F.

`Cone(F, i): TorFan, RngIntElt -> TorCon`

The ith cone of the sequence of maximal cones of the fan F.

`Cone(F, S): TorFan, [RngIntElt] -> TorCon`

```
in_fan: BoolElt                    Default: true
extend: BoolElt                    Default: false
```

The cone C spanned by those rays in the fan F whose indices appear in S. By default C must be a cone in F. If the optional parameter `extend` is set to `true` then the smallest cone in F containing C will be found (if such a cone exists). If the parameter `in_fan` is set to `false` then the cone is forcibly created even if it failed previous tests.

`NonSimplicialCones(F): TorFan -> SeqEnum, SeqEnum`

The minimum sequence of non-simplicial (i.e. non-$\mathbb{Q}$-factorial) cones of the fan F. Also gives the sequences of indices of the rays of F defining each non-simplicial cone.

`SingularCones(F): TorFan -> SeqEnum, SeqEnum`

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: Singular Cones of a Weighted Projective Space `(ex-e75704)`

The weighted projective space $\mathbb{P}(1,2,2,3)$ has a singular line (with stabiliser $\mathbb{Z}/2$) and a singular point (with stabiliser $\mathbb{Z}/3$).

```
> 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.

`ConeIndices(F): TorFan -> SeqEnum`

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): TorFan, TorCon -> SeqEnum`

The sequence of integers that are the indices of the rays which generate the cone C of the fan F.

`ConeIntersection(F, C1, C2): TorFan, TorCon, TorCon -> TorCon`

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): TorFan, TorCon -> TorCon`

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.)

`InnerNormals(F): TorFan -> SeqEnum`

`OuterNormals(F): TorFan -> SeqEnum`

If all the cones of the fan F are $\mathbb{Q}$-Gorenstein, this returns a sequence of elements in the dual lattice where the i-th element is the inner (resp. outer) normal of the i-th cone of F.

`DualFaceInDualFan(P, Q): TorPol, [RngIntElt] -> TorFan`

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): TorFan -> SeqEnum`

A sequence containing the rays of the fan F (as a sequence of primitive lattice points on each ray).

`Ray(F, i): TorFan, RngIntElt -> TorLatElt`

The ith ray of the fan F (regarded as the primitive ambient toric lattice point on the ray).

`AllRays(F): TorFan -> SeqEnum`

A sequence of the rays of the fan F (including all virtual rays).

`PureRays(F): TorFan -> SeqEnum`

A sequence of the (non-virtual) rays of the fan F.

`PureRayIndices(F): TorFan -> SeqEnum`

A sequence of the indices of the non-virtual rays of the fan F among all its rays.

`CreateVirtualRays(S): [TorLatElt] -> SeqEnum`

Given a sequence S of rays of a fan, computes the remaining virtual rays which form a basis of the lattice complementary to the linear span of S. Virtual rays are useful, for example, when calculating the coordinate ring of the toric variety associated to the fan when the variety is a product of a torus and a smaller toric variety.

`VirtualRays(F): TorFan -> SeqEnum`

A sequence of the virtual rays of the fan F.

`VirtualRayIndices(F): TorFan -> SeqEnum`

A sequence of the indices of the virtual rays of the fan F among all its rays.

129.2.3 Properties of Fans

`Ambient(F): TorFan -> TorLat`

The ambient toric lattice of the toric fan F.

`IsComplete(F): TorFan -> BoolElt`

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): TorFan -> BoolElt`

Return `false` if and only if all the cones of the fan F are nonsingular.

`IsNonsingular(F): TorFan -> BoolElt`

Return `true` if and only if all the cones of the fan F are nonsingular.

`IsQFactorial(F): TorFan -> BoolElt`

Return `true` if and only if all the cones of the fan F are $\mathbb{Q}$-factorial.

`IsIsolated(F): TorFan -> BoolElt`

Return `true` if and only if all the cones of the fan F are isolated.

`IsTerminal(F): TorFan -> BoolElt`

Return `true` if and only if all the cones of the fan F are terminal.

`IsCanonical(F): TorFan -> BoolElt`

Return `true` if and only if all the cones of the fan F are canonical.

`IsGorenstein(F): TorFan -> BoolElt`

Return `true` if and only if all the cones of the fan F are Gorenstein.

`IsQGorenstein(F): TorFan -> BoolElt`

Return `true` if and only if all the cones of the fan F are $\mathbb{Q}$-Gorenstein.

129.2.4 Maps of Fans

`F @ f: TorFan, Map -> TorFan`

The image of the fan F by the map f of toric lattices.

`SimplicialSubdivision(F): TorFan -> TorFan`

`SimplicialSubdivision(C): TorCon -> TorFan`

A toric fan that is a simplicial subdivision of the toric fan F (or of the toric cone C).

Example: Simplicial Subdivision of a Cone `(ex-8ec30f)`

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): TorFan, TorFan -> BoolElt`

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): TorFan, TorFan, Map -> BoolElt`

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): TorFan, TorFan -> TorFan`

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.)

`Resolution(F): TorFan -> TorFan`

`deterministic: BoolElt`	`Default: true`

A resolution of singularities of the fan F. If the optional parameter `deterministic` is set to `false` then a potentially much faster non-deterministic algorithm will be used to calculate the resolution.

`Terminalisation(F): TorFan -> TorFan`

A $\mathbb{Q}$-factorial terminal refinement of the fan F.

`Canonicalisation(F): TorFan -> TorFan`

A $\mathbb{Q}$-factorial canonical refinement of the fan F.

129.3 Geometrical Properties of Cones and Polyhedra

`IsSingular(C): TorCon -> BoolElt`

Return `true` if and only if the affine variety associated with the cone C is singular.

`IsNonsingular(C): TorCon -> BoolElt`

Return `true` if and only if the affine variety associated with the cone C is nonsingular.

`IsSmooth(P): TorPol -> BoolElt`

Return `true` if and only if the polyhedron P is a smooth polytope.

`IsGorenstein(C): TorCon -> BoolElt`

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): TorPol -> BoolElt`

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): TorCon -> BoolElt`

Return `true` if and only if the cone C has (the primitive points on its) rays contained in an affine hyperplane.

`GorensteinIndex(C): TorCon -> RngIntElt, TorLatElt`

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 $\mathbb{Q}$-Gorenstein.)

`GorensteinIndex(P): TorPol -> RngIntElt`

The Gorenstein index of the lattice polytope P; i.e. the smallest positive integer k such that $kP^\vee$ is an integral polytope.

`IsIsolated(C): TorCon -> BoolElt`

Return `true` if and only if the singularity of the affine variety associated to the cone C is isolated.

`IsQFactorial(C): TorCon -> BoolElt`

`IsSimplicial(P): TorPol -> BoolElt`

Return `true` if and only if the cone C or polytope P is simplicial.

`IsTerminal(C): TorCon -> BoolElt`

Return `true` if and only if the singularity of the affine variety associated to the cone C is (at worst) terminal.

`IsCanonical(C): TorCon -> BoolElt`

Return `true` if and only if the singularity of the affine variety associated to the cone C is (at worst) canonical.

`IsFano(P): TorPol -> BoolElt`

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: Cone of a Terminal Quotient Singularity (ex-ee8303)

We make the cone corresponding to the (affine) terminal quotient singularity $\mathbb{C}^3/(\mathbb{Z}/5)$ where $\mathbb{Z}/5$ acts as the 5th roots of unity in the diagonal representation $\mathrm{diag}(1,2,3)$.

```
> 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 $1/5(4,3,2)$.

129.4 Toric Varieties

129.4.1 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): Fld, RngIntElt -> TorVar

Projective n-space $\mathbb{P}^n$ defined over the field k as a toric variety.

ToricVariety(k, Z): Fld, [RngIntElt] -> TorVar

The (weighted) projective space $\mathbb{P}(Z)$ defined over the field k with weights the positive integer sequence Z as a toric variety.

ToricVariety(k, Z, Q): Fld, [RngIntElt], [FldRatElt] -> TorVar

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): Fld, [[RngIntElt]], [RngIntElt] -> TorVar

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: Hirzebruch Surface as a GIT Quotient (ex-70f3b4)

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): Fld -> TorVar

The zero-dimensional point over the field k defined as a toric variety.

ProjectiveSpace(k, n): Fld, RngIntElt -> Prj
ProjectiveSpaceAsToricVariety(k, n): Fld, RngIntElt -> TorVar
WeightedProjectiveSpace(k, n): Fld, RngIntElt -> TorVar
WPS(k, n): Fld, RngIntElt -> TorVar

Projective n-space $\mathbb{P}^n$ defined over the field k.

ProjectiveSpace(k, W): Fld, [RngIntElt] -> Prj
ProjectiveSpaceAsToricVariety(k, W): Fld, [RngIntElt] -> TorVar
WeightedProjectiveSpace(k, W): Fld, [RngIntElt] -> TorVar
WPS(k, W): Fld, [RngIntElt] -> TorVar
FakeProjectiveSpace(k, W, Q): Fld, [RngIntElt], [FldRatElt] -> TorVar

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 Q, if provided).

`AbsoluteRationalScroll(k, S): Fld, [RngIntElt] -> TorVar`

Given a sequence of weights $S = (s_1, \ldots, s_n)$ of length n, creates the n-dimensional rational scroll over k with gradings $(1, 1, -s_1, \ldots, -s_n), (0, 0, 1, \ldots, 1)$.

`RationalScroll(k, s, A): Fld, RngIntElt, [RngIntElt] -> TorVar`

The fibre bundle of projective r-spaces over projective s-space, where $A = [a_0, ..., a_r]$ is a sequence of non-negative integers of length at least 2, that is $\mathrm{Proj}(O(a_0) \oplus \ldots \oplus O(a_r))$.

`RuledSurface(k, n): Fld, RngIntElt -> TorVar`

The rational ruled Hirzebruck surface of degree $n \geq 0$ over k. This is the rational ruled surface with special section of self-intersection $-n$. It is the toric variety having the gradings $(1, 1, -n, 0)$ and $(0, 0, 1, 1)$ on four variables.

`RuledSurface(k, a1, a2): Fld, RngIntElt, RngIntElt -> TorVar`

The rational ruled surface with gradings $(1, 1, -a1, -a2)$ and $(0, 0, 1, 1)$ on four variables. This is the ruled surface $P(O(-a1) \oplus O(-a2))$ in Hartshorne's notation (Section 2, Chapter V of [Har77]). It is abstractly isomorphic to the Hirzebruch surface of degree $|a1 - a2|$.

`HirzebruchSurface(k, n): Fld, RngIntElt -> TorVar`

The same as `RuledSurface(k,n)` except that the last two variables are swapped, ie the gradings are $(1, 1, 0, -n)$ and $(0, 0, 1, 1)$.

`BigTorus(k, N): Rng, TorLat -> TorVar`

The big torus T associated with a toric lattice N.

`BigTorus(X): TorVar -> TorVar, TorMap, TorMap`

The big torus T of the toric variety X, together with its embedding into X and the rational map from X to T.

`RestrictionToSubtorus(Z): Sch -> Sch, TorMap`

The restriction of Z to the largest subtorus of the ambient containing Z.

129.4.2 Toric Varieties and their Fans

`ToricVariety(k, F): Fld, TorFan -> TorVar`

The toric variety (defined over the field k) corresponding to the toric fan F.

`Fan(X): TorVar -> TorLat`

The toric fan corresponding to the toric variety X.

`Rays(X): TorVar -> SeqEnum`

The rays of the fan of the toric variety X.

`OneParameterSubgroupsLattice(X): TorVar -> TorLat`

The lattice of weights of the toric variety X; this is the lattice which supports the toric fan of X.

`MonomialLattice(X): TorVar -> TorLat`

The monomial lattice of the toric variety X, namely the toric lattice dual to that containing the fan of X.

`CoxMonomialLattice(X): TorVar -> TorLat`

The lattice whose elements represent Weil divisors on the toric variety X; it is dual to ray lattice of X.

`PicardLattice(X): TorVar -> TorLat`

The lattice whose integral elements correspond to Cartier divisors up to linear equivalence and modulo torsion.

`DivisorClassLattice(X): TorVar -> TorLat`

The divisor class lattice of the toric variety X.

`IrrelevantIdeal(X): TorVar -> SeqEnum`

A sequence of ideals that are the components of the irrelevant ideal of the toric variety X.

`Gradings(X): Sch -> SeqEnum`

A sequence of sequences representing all of the gradings to be applied to the indeterminates of the coordinate ring of the ambient of X.

`NumberOfGradings(X): Sch -> RngIntElt`

The number of gradings to be applied to the indeterminates of the coordinate ring of the ambient of X.

`QuotientGradings(X): TorVar -> SeqEnum`

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): TorVar -> RngIntElt`

The number of sequences the generate the quotient gradings of the toric variety X.

129.4.3 Properties of Toric Varieties

`IsSingular(X): TorVar -> BoolElt`

Return `false` if and only if the toric variety X is nonsingular.

`IsNonsingular(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X is nonsingular.

`IsGorenstein(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X is Gorenstein.

`IsQGorenstein(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X is $\mathbb{Q}$-Gorenstein.

`IsQFactorial(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X is $\mathbb{Q}$-factorial.

`IsIsolated(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X has only isolated singularities.

`IsTerminal(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X has (at worst) terminal singularities.

`IsCanonical(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X has (at worst) canonical singularities.

`IsComplete(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X is complete.

`IsProjective(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X is projective.

`IsFano(X): TorVar -> BoolElt`

Return `true` if and only if the anticanonical divisor of the toric variety X is ample.

`IsWeakFano(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X is a weak Fano variety.

`IsFakeWeightedProjectiveSpace(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X has exactly one $\mathbb{Z}$-grading.

`IsWeightedProjectiveSpace(X): TorVar -> BoolElt`

Return `true` if and only if the toric variety X has exactly one $\mathbb{Z}$-grading and no quotient gradings.

129.4.4 Affine Patches on Toric Varieties

`ToricAffinePatch(X, i): TorVar, RngIntElt -> TorVar, TorMap`

The affine patch corresponding to i-th cone of fan of the toric variety X together with the inclusion map.

`ToricAffinePatch(X, S): TorVar, [RngIntElt] -> TorVar, TorMap`

`ToricAffinePatch(X, S): TorVar, [RngMPolElt] -> TorVar, TorMap`

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.

129.5 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 $\boldsymbol{Z}$ *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.

129.5.1 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): TorVar -> RngCox`

The Cox ring of the toric variety X.

`CoxRing(k, F): Fld, TorFan -> RngCox`

The Cox ring of the toric variety defined over field k by the fan F.

Example: Cox Ring of a Weighted Projective Plane (ex-d5e182)

We build the weighted projective space $\mathbb{P}^2(1,2,3)$.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),[1,2,3]);
```

The Cox ring of $\mathbb{P}^2(1,2,3)$ 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 $\mathbb{P}^2$ since it lies in the locus defined by the irrelevant ideal.

Example: Toric Variety with a Fan Like the Plane (ex-7eeae1)

We build a toric variety X_2 whose fan resembles that of $\mathbb{P}^2$.

```
> 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 $\mathbb{P}^2$. 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 $\mathbb{P}^2$ 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 $\mathbb{P}^2$. 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 $\mathbb{P}^2$ by the action of $\mathbb{Z}/3$ given by

$$\varepsilon\colon (u,v,w) \mapsto (u, \varepsilon^2 v, \varepsilon w).$$

where ε is a cube-root of unity.

129.5.2 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): RngMPol, SeqEnum, SeqEnum, SeqEnum -> RngCox

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: RngCox, RngCox -> BoolElt

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): RngCox -> Fld

CoefficientRing(C): RngCox -> Fld

The coefficient field of the Cox ring C.

UnderlyingRing(C): RngCox -> RngMPol

The underlying polynomial ring of the Cox ring C.

Length(C): RngCox -> RngIntElt

The rank of the underlying polynomial ring of the Cox ring C, that is, the number of polynomial variables of C.

IrrelevantIdeal(C): RngCox -> SeqEnum

The irrelevant ideal of the Cox ring C.

`IrrelevantComponents(C): RngCox -> SeqEnum`

A sequence containing the components of the irrelevant ideal of the Cox ring C.

`IrrelevantGenerators(C): RngCox -> SeqEnum`

A sequence of sequences, each containing the generators of the components of the irrelevant ideal of the Cox ring C.

`Gradings(C): RngCox -> RngIntElt`

The $\mathbf{Z}$ gradings of the Cox ring C, that is, a sequence of sequences of integers.

`NumberOfGradings(C): RngCox -> RngIntElt`

The number of $\mathbf{Z}$ gradings of the Cox ring C.

`QuotientGradings(C): RngCox -> RngIntElt`

The quotient gradings of the Cox ring C, that is, a sequence of sequences of rational numbers.

`NumberOfQuotientGradings(C): RngCox -> RngIntElt`

The number of quotient gradings of the Cox ring C.

`C . i: RngCox, RngInt -> RngMPolElt`

The i-th indeterminate for the underlying polynomial ring of the Cox ring C.

`AssignNames(~C, S): RngCox, [MonStgElt]`

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): RngCox, RngIntElt -> RngMPolElt`

The ith variable of the underlying polynomial ring of the Cox ring C.

129.5.3 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 $\mathbf{Z}$ 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): RngCox -> TorVar`

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: Toric Variety from a Cox Ring `(ex-e56de4)`

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:
```

(continues on next page)

(continued from previous page)

```
    (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): RngCox -> TorFan

The fan associated to the Cox ring is C; an error is reported if there is no such fan.

CoxMonomialLattice(C): RngCox -> TorLat

The Cox monomial lattice of the Cox ring C.

BasisOfRationalFunctionField(X): TorVar -> SeqEnum

A basis of the rational function field of the toric variety X expressed in terms of rational monomials in the Cox ring.

BasisOfDegree0CoxMonomials(X): TorVar -> SeqEnum

A basis of the rational function field of the toric variety X expressed in terms of lattice points in the Cox monomials lattice.

DivisorClassLattice(C): RngCox -> TorLat

The divisor class lattice of the Cox ring C.

MonomialLattice(C): RngCox -> TorLat

The monomial lattice of the Cox ring C.

OneParameterSubgroupsLattice(C): RngCox -> TorLat

The one-parameter subgroups lattice of the Cox ring C.

RayLattice(C): RngCox -> TorLat

The ray lattice of the Cox ring C.

DivisorClassGroup(C): RngCox -> TorLat

The divisor class group of the Cox ring C.

RayLatticeMap(C): RngCox -> Map

The map from the ray lattice of the Cox ring C to the ambient lattice of its fan.

WeilToClassGroupsMap(C): RngCox -> Map

`WeilToClassLatticesMap(C): RngCox -> Map`

Comparison maps between toric lattices related to the Cox ring C of a toric variety.

129.6 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.)

129.6.1 Divisor Group

`DivisorGroup(X): TorVar -> DivTor`

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): DivTor -> TorVar`

The toric variety of which G is the divisor group.

`G1 eq G2: DivTor, DivTor -> BoolElt`

Return `true` if and only if the divisor groups G_1 and G_2 are those of the same toric variety.

`Divisor(G, S): DivTor, [RngIntElt] -> DivTorElt`

`Divisor(G, S): DivTor, [FldRatElt] -> DivTorElt`

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): DivTor, RngIntElt -> DivTorElt`

The divisor on the toric variety X associated to the divisor group G given by the vanishing of the ith coordinate of X.

129.6.2 Constructing Invariant Divisors

`Divisor(X, S): TorVar, [RngIntElt] -> DivTorElt`

The Weil divisor (respectively $\mathbb{Q}$-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): TorVar, RngIntElt -> DivTorElt`

The divisor on the toric variety X given by the vanishing of the ith coordinate of X.

`Divisor(X, f): TorVar, RngMPolElt -> DivTorElt`

The divisor on the toric variety X defined by the polynomial f of the Cox ring of X.

`Divisor(X, m): TorVar, TorLatElt -> DivTorElt`

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): TorVar -> DivTorElt`

The zero divisor on the toric variety X.

Representative(X, m): TorVar, ModEDElt -> DivTorElt

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): TorVar, TorLatElt -> DivTorElt

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): TorVar -> DivTorElt

The canonical divisor of the toric variety X.

CanonicalClass(X): TorVar -> DivTorElt

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 $\mathbb{Q}$-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: DivTorElt, DivTorElt -> DivTorElt

n * D: FldRatElt, DivTorElt -> DivTorElt

- D: DivTorElt -> DivTorElt

D1 - D2: DivTorElt, DivTorElt -> DivTorElt

D * v: DivTorElt, TorLatElt -> RngIntElt

Standard arithmetic operations for divisors D, D_1, D_2 on a toric variety, where $n \in \mathbb{Q}$ and v is a point of the ambient toric lattice of the corresponding fan.

Example: Kawamata Blowup (ex-9d2674)

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
```

129.6.3 Properties of Divisors

Variety(D): DivTorElt -> TorVar

The toric variety on which the divisor D is defined.

Parent(D): DivTorElt -> DivTor

The divisor group of a toric variety in which the divisor D lies.

Weil(D): DivTorElt -> SeqEnum

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): DivTorElt -> SeqEnum[TorLatElt]

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 $\mathbb{Q}$-Cartier.

IsQCartier(D): DivTorElt -> BoolElt

Return **true** if and only if some integer multiple of the divisor D on a toric variety is Cartier.

IsCartier(D): DivTorElt -> BoolElt

Return **true** if and only if the divisor D on a toric variety is Cartier.

IsWeil(D): DivTorElt -> BoolElt

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): DivTorElt -> BoolElt

Returns **true** if and only if the divisor D on a toric variety is ample.

IsNef(D): DivTorElt -> BoolElt

Return **true** if and only if the divisor D on a toric variety is nef.

IsBig(D): DivTorElt -> BoolElt

Return **true** if and only if the divisor D on a toric variety is big.

IsEffective(D): DivTorElt -> BoolElt

Return **true** if and only if the divisor D on a toric variety is effective.

PicardClass(D): DivTorElt -> TorLatElt

The class in the Picard lattice corresponding to the $\mathbb{Q}$-Cartier divisor D.

`MovablePart(D): DivTorElt -> DivTorElt`

The movable part (or mobile part) of the divisor D on a toric variety.

Example: Movable Divisors on a Blowup (ex-852f90)

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): DivTorElt -> TorFan`

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): DivTorElt -> TorVar, PlcEnum`

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)` $\to$ `Proj(D)` is also returned.

`RelativeProj(D): DivTorElt -> TorVar`

The relative (sheaf) Proj of sections of the divisor D on a toric variety. If D is $\mathbb{Q}$-Cartier, then the identity will be constructed; for non $\mathbb{Q}$-Cartier divisors, a partial $\mathbb{Q}$-factorialisation will be given.

`IntersectionForm(X, C): TorVar, TorCon -> TorLatElt`

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.

`IntersectionForms(X): TorVar -> [TorLatElt]`

The sequence of forms on the Picard lattice of the toric variety X which define the intersection forms with all torus invariant curves.

`CartierToWeilMap(X): TorVar -> Map`

The embedding map from the lattice of torus-invariant Cartier divisors to the lattice of torus-invariant Weil divisors of the toric variety X.

`PicardToClassGroupsMap(X): TorVar -> Map`

The embedding of the Picard group of X into the divisor class group.

`PicardToClassLatticesMap(X): TorVar -> Map`

The natural embedding of the Picard lattice of X into divisor class lattice.

129.6.4 Linear Equivalence of Divisors

`IsQPrincipal(D): DivTorElt -> BoolElt`

Return `true` if and only if some integer multiple of the divisor D on a toric variety is principal.

`IsPrincipal(D): DivTorElt -> BoolElt`

Return `true` if and only if the divisor D on a toric variety is principal.

`IsLinearlyEquivalentToCartier(D): DivTorElt -> BoolElt, DivTorElt`

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): DivTorElt, DivTorElt -> BoolElt`

`IsLinearlyEquivalent(D, E): DivTorElt, DivTorElt -> BoolElt`

Return `true` if and only if the divisors D and E are linearly equivalent.

`LinearlyEquivalentDivisorWithNoSupportOn(D, S): DivTorElt, [RngMPolElt] -> DivTorElt`

A divisor linearly equivalent to the divisor D, with no support on the locus defined by S. S should be a sequence of variables on the variety of D. If S has more than just variables, the other polynomials will be ignored.

`DefiningMonomial(D): DivTorElt -> RngMPolElt`

The monomial (if D is effective) or rational monomial defining the divisor D on a toric variety.

`LatticeElementToMonomial(D, v): DivTorElt, TorLatElt -> RngMPolElt`

The monomial in the Cox ring that corresponds to the monomial lattice element v when regarded as a section of the divisor D.

129.6.5 Riemann–Roch Spaces of Invariant Divisors

`RiemannRochPolytope(D): DivTorElt -> TorPol`

The Riemann–Roch space of the divisor D as a polytope in the monomial lattice of the underlying toric variety.

`RiemannRochBasis(D): DivTorElt -> [RngElt]`

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): DivTorElt -> RngIntElt`

The dimension of the Riemann–Roch space of the divisor D on a toric variety.

`GradedCone(D): DivTorElt -> TorCon`

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): DivTorElt -> TorPol`

The integral polyhedron whose integral points corresponds to sections of the divisor D.

Example: Riemann-Roch Space on a Blown-Up Plane (ex-35b77b)

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): DivTor -> FldFunRatUElt

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): DivTor -> [RngUPolElt]

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): DivTor, RngIntElt -> [RngIntElt]

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): DivTor, RngIntElt -> RngIntElt

Thefirst ith coefficient of the Hilbert series of the divisor D on a toric variety.

HilbertDeltaVector(D): DivTor -> [RngIntElt]

The Hilbert δ-vector (or h^*-vector) for the divisor D on a toric variety.

Example: Riemann-Roch Space by Cech Cohomology (ex-ad5b0b)

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
```

129.7 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 $\mathbb{P}^1$ to $\mathbb{P}(1, 2, 3)$, 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 $\mathbb{P}(1,2,3)$ 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 $\mathbb{P}^1$.

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.

129.7.1 Maps from Lattice Maps

`ToricVarietyMap(X, Y, f): TorVar, TorVar, Map -> TorMap`

`ToricVarietyMap(X, Y): TorVar, TorVar -> TorMap`

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): TorVar, TorLatElt -> TorVar, TorMap`

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.

`ToricIdentityMap(X): TorVar -> TorMap`

The identity map on the toric variety X.

129.7.2 Properties of Toric Maps

`IsRegular(f): TorMap -> BoolElt`

Return `true` if and only if the map f between toric varieties is regular.

`IndeterminacyLocus(f): TorMap -> [Sch]`

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: Map from a Hirzebruch Surface `(ex-cf6855)`

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 $\mathbb{P}(1,1,2)$. 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:
```

(continues on next page)

(continued from previous page)

```
    1,
    (v)*(u)^(-1),
    (x)*(y)^(-1)*(u)^(-2)
```

Now we can ask whether this map F is a morphism.

```
> IsRegular(F);
true
```

129.8 The Geometry of Toric Varieties

129.8.1 Resolution of Singularities and Linear Systems

`Resolution(X): TorVar -> TorVar, TorMap`

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.

`QFactorialisation(X): TorVar -> TorVar, TorMap`

A $\mathbb{Q}$-factorialisation of X together with the natural morphism.

`Terminalisation(X): TorVar -> TorVar, TorMap`

A $\mathbb{Q}$-factorial terminal blowup of the toric variety X and the natural morphism.

`Canonicalisation(X): TorVar -> TorVar, TorMap`

A $\mathbb{Q}$-factorial canonical blowup of the toric variety X and the natural morphism.

`ResolveLinearSystem(D): DivTorElt -> TorVar`

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.

129.8.2 Mori Theory of Toric Varieties

`MoriCone(X): TorVar -> TorCon`

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): TorVar -> TorCon`

The nef Cone of toric variety X (as an abstract cone).

`ExtremalRays(X): TorVar -> SeqEnum`

`ExtremalRayContractions(X): TorVar, RngIntElt -> SeqEnum`

The images of extremal contractions of rays in the nef-cone of the toric variety X.

`ExtremalRayContraction(X, i): TorVar, RngIntElt -> TorVar, TorMap`

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): TorVar, RngIntElt -> DivTorElt`

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): TorVar, RngIntElt -> MonStgElt`

`TypesOfContractions(X): TorVar -> MonStgElt`

A string describing the ith extremal contraction of the toric variety X (or all together in a sequence if i is not specified).

`IsDivisorialContraction(X, i): TorVar, RngIntElt -> BoolElt`

True iff the contraction of the ith extremal ray of the toric variety X is a divisorial contraction.

`IsMoriFibreSpace(X, i): TorVar, RngIntElt -> BoolElt`

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.

`IsFlipping(X, i): TorVar, RngIntElt -> BoolElt`

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): TorVar, RngIntElt -> TorVar`

The flipped variety of the ith extremal contraction of the toric variety X, assuming that this extremal contraction is of flipping type.

`Flip(D): DivTorElt -> TorVar`

The (generalised) flip of the morphism given by the $\mathbb{Q}$-Cartier divisor, assuming that this morphism is small.

`WeightsOfFlip(X, i): TorVar, RngIntElt -> SeqEnum`

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: Extremal Contractions of a Blowup (ex-33c790)

```
> 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: Weights of a Flip (ex-ddd3e8)

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 $\mathbb{P}^1$.

```
> 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 $\mathbb{P}^1$. 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 $\mathbb{P}^1$ has been antiflipped in favour of a weighted $\mathbb{P}^4(1, 1, 1, 1, 2)$, which is the source of the singularity on Y.

`MMP(X): TorVar -> SeqEnum, SeqEnum`

```
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: Minimal Model Program on a Blowup `(ex-cb6736)`

First make a toric variety X, in this case some blowup of $\mathbb{P}^3$.

```
> 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) $\mathbb{P}^3$, and (3) a point. We also see two maps: the first, labelled $[1, 2]$, is the contraction of X back down to $\mathbb{P}^3$, and the second is the extremal contraction of $\mathbb{P}^3$ 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:
```

(continues on next page)

(continued from previous page)

```
        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 $\mathbb{P}^1$ (which we could check is model number 5).

129.8.3 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: Relative Minimal and Canonical Models `(ex-444c76)`

We will compute a relative minimal model and a relative canonical model of the weighted projective space $\mathbb{P}^3(1,2,5,6)$, 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,
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
> 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.

129.9 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.

129.9.1 Construction of Subschemes

`Scheme(X, f): TorVar, RngMPolElt -> Sch`

The subscheme of the toric variety X defined by the polynomial f from the Cox ring of X.

`Scheme(X, Q): TorVar, [RngMPolElt] -> Sch`

The subscheme of the toric variety X defined by the sequence Q of polynomials from the Cox ring of X.

`BinomialToricEmbedding(Z): Sch -> Sch, TorMap`

Takes the binomial equations in the ideal of the scheme Z and constructs the toric variety given by the normalisation of the closure of the subtorus described by those binomials. Returns the pullback of Z and the normalisation map into the ambient of Z.

Example: Trigonal Curve in a Toric Variety (ex-641627)

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 $X = \mathbb{P}^7$. We can recover this toric variety and its map to X.

```
> W,g := BinomialToricEmbedding(V);
> Y<[b]> := Domain(g);
> Y;
```

(continues on next page)

(continued from previous page)

```
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 $\mathbb{P}^1$, 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.

129.10 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. ISBN 0-691-00049-2. The William H. Roever Lectures in Geometry.
- [Har77] Robin Hartshorne. *Algebraic Geometry, GTM 52.* Springer, ASpringer, 1977.
- [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. ISBN 3-540-17600-4. 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.

References

[Alt98] Selma Altnok. *Graded Rings Corresponding to Polarised K3 Surfaces and Q-Fano 3-folds*. PhD thesis, University of Warwick, 1998. http://www.maths.warwick.ac.uk/ miles/students/Selma/.

[ABR02] Selma Altnok, 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.

[AGZV85] V. I. Arnold, S. M. Gusein-Zade, and A. N. Varchenko. *Singularities of Differentiable Maps Volume I*. Volume 82 of Monographs in Mathematics. Birkhauser, 1985.

[BHPdV04] Barth, Hulek, Peters, and Van de Ven. *Compact Complex Surfaces*. Ergebnisse der Mathematik und ihrer Grenzgebiete 4. Springer, second edition, 2004. ISBN 3-540-00832-2.

[Bec07] Tobias Beck. Formal Desingularization of Surfaces – The Jung Method Revisited –. Technical Report 2007-31, RICAM, December 2007. http://www.ricam.oeaw.ac.at/publications/reports/.

[Bec08] Tobias Beck. Software Documentation. Technical Report 2008-8, RICAM, May 2008. http://www.ricam.oeaw.ac.at/publications/reports/.

[BS08] Tobias Beck and Josef Schicho. Adjoint Computation for Hypersurfaces Using Formal Desingularizations. Technical Report 2008-2, RICAM, January 2008. http://www.ricam.oeaw.ac.at/publications/reports/.

[BFHN] M. Besier, D. Festi, M.C. Harrison, and B. Naskrecki. Arithmetic and geometry of a K3 surface emerging from virtual corrections to Drell-Yan scattering. preprint arXiv:1908.01079 [math.AG].

[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.).

[Bru] N. Bruin. "Some ternary Diophantine equations of signature $(n, n, 2)$."

[Buc03] Anita Buckley. *Orbifold Riemann-Roch for 3-folds and applications to Calabi-Yaus*. PhD Thesis, Warwick University, 2003.

[CT17] Wouter Castryck and Jan Tuitman. Point counting on curves using a gonality preserving lift. *The Quarterly Journal of Mathematics*, 2017.

[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.

[dGP] W.A. de Graaf and J. Pilnikova. Parametrizing Del Pezzo surfaces of degree 8 using Lie algebras. http://arxiv.org/abs/math.NT/0512477.

[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.

[DES93] Decker, Ein, and Schreyer. Construction of surfaces in P4. *J. Algebraic Geometry*, 2:185–237, 1993.

[Die] C. Diem. "An Index Calculus Algorithm for Plane Curves of Small Degree." 543–557.

[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. ISBN 0-691-08380-0.

[Eis95] David Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry*. Volume 150 of Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1995.

[Eis05] David Eisenbud. *The geometry of syzygies: a second course in commutative algebra and algebraic geometry*. Volume 225 of Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2005.

[Elk] N. Elkies. "Rational Points Near Curves and Small Nonzero $|x^3-y^2|$ via Lattice Reduction." 33–63.

[Els] Andreas-Stephan Elsenhans. Good models for cubic surfaces. (To appear). URL: http://www.staff.uni-bayreuth.de/ btm216/.

[EJ10a] Andreas-Stephan Elsenhans and Jörg Jahnel. Cubic surfaces with a Galois invariant double-six. *Cent. Eur. J. Math.*, 8(4):646–661, 2010.

[EJ10b] Andreas-Stephan Elsenhans and Jörg Jahnel. Weil polynomials of K3 surfaces. In *Algorithmic number theory*, volume 6197 of Lecture Notes in Computer Science, 126–141. Berlin, 2010. Springer.

[Fes] Dino Festi. A practical algorithm to compute the geometric picard lattices of K3 surfaces of degree 2. preprint arXiv:1808.00351 [math.AG].

[Ful69] William Fulton. *Algebraic Curves*. Volume of Mathematics Lecture Note Series. W. A. Benjamin, New York-Amsterdam, 1969.

[Ful93] William Fulton. *Introduction to toric varieties*. Volume 131 of Annals of Mathematics Studies. Princeton University Press, Princeton, NJ, 1993. ISBN 0-691-00049-2. The William H. Roever Lectures in Geometry.

[GSHPBS12] J. Gonzalez-Sanchez, M. Harrison, I. Polo-Blanco, J. Schicho. "Algorithms For Del Pezzo Surfaces of Degree 5 (Construction, Parametrization)." JSC, 147(3), 342–353. 2012.

[Har13] M. C. Harrison. Explicit solution by radicals, gonal maps and plane models of algebraic curves of genus 5 or 6. *J. Symb. Comp.*, 51:3–21, 2013.

[HS06] M.C. Harrison and J. Schicho. Rational Parametrisation for Degree 6 Del Pezzo Surfaces using Lie Algebras. In *Proceedings ISSAC'06*. 2006.

[Har12] Michael C. Harrison. An extension of Kedlaya's algorithm for hyperelliptic curves. *J. Symbolic Comput.*, 47(1):89–101, 2012.

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

[Har15] David Harvey. Computing zeta functions of arithmetic schemes. *Proceedings of the London Mathematical Society*, 111(6):1379–1401, 2015. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/pdv056, arXiv:https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/pdv056, doi:https://doi.org/10.1112/plms/pdv056.

[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.

[Hor76] E. Horikawa. Algebraic Surfaces of General Type with Small c_1^2. II. *Invent. Math.*, 37:121–155, 1976.

[HM73] G. Horrocks and D. Mumford. A rank 2 vector bundle on $\mathbf{P}_4$ with 15,000 symmetries. *Topology*, 12:63–81, 1973.

[Hun96] Bruce Hunt. *The geometry of some special arithmetic quotients*. Volume 1637 of Lecture Notes in Mathematics. Springer-Verlag, Berlin, 1996.

[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.

[KS00b] Maximilian Kreuzer and Harald Skarke. Complete classification of reflexive polyhedra in four dimensions. *Adv. Theor. Math. Phys.*, 4(6):1209–1230, 2000.

[Kyn22] Madeleine Kyng. Computing zeta functions of algebraic curves using Harvey's trace formula. *Res. Number Theory*, 8(4):Paper No. 100, 17, 2022. URL: https://doi.org/10.1007/s40993-022-00398-7, doi:10.1007/s40993-022-00398-7.

[Kyn25] Madeleine Kyng. *Computing zeta functions of algebraic curves using Harvey's trace formula*. Dissertation, The University of New South Wales, May 2025. available at https://web.maths.unsw.edu.au/ davidharvey/research/kyng-thesis.pdf.

[Man86] Yu. I. Manin. *Cubic Forms (2nd ed.)*. North-Holland Publishing Co., Amsterdam, 1986. Vol. 4 of North-Holland Mathematical Library.

[Mat89] H. Matsumura. *Commutative ring theory*. CUP, ACUP, 1989.

[MSS96] J. R. Merriman, S. Siksek, and N. P. Smart. Explicit 4-descents on an elliptic curve. *Acta Arith.*, 77(4):385–404, 1996.

[Min13] Moritz Minzlaff. *Frobenius-stable lattices in rigid cohomology of curves*. Dissertation, Technische Universität Berlin, 2013. URL: http://dx.doi.org/10.14279/depositonce-3566.

[MN17] Pascal Molin and Christian Neurohr. Computing period matrices and the Abel-Jacobi map of superelliptic curves. *Mathematics of Computation*, Dec 2017. URL: http://dx.doi.org/10.1090/mcom/3351, doi:10.1090/mcom/3351.

[Neu99] W.D. Neumann. Irregular links at infinity of complex affine plane curves. *Quart. J. Math. Ox. Ser. (2)*, 50:301–320, 1999.

[Neu18] Christian Neurohr. *Efficient integration on Riemann surfaces & applications*. Dissertation, Carl von Ossietzky Universität Oldenburg, 2018. URL: http://oops.uni-oldenburg.de/3607.

[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. ISBN 3-540-17600-4. An introduction to the theory of toric varieties, Translated from the Japanese.

[Pap03] S. Papadakis. Kustin–Miller unprojection with complexes. *J. Algebraic Geometry*, 2003.

[RS00] Kristian Ranestad and Frank-Olaf Schreyer. Varieties of sums of powers. *J. Reine Angew. Math.*, 525:147–181, 2000. URL: http://dx.doi.org/10.1515/crll.2000.064, doi:10.1515/crll.2000.064.

[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.

[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.

[Rei00] Miles Reid. Graded rings and birational geometry. In K. Ohno, editor, *Proc. of algebraic geometry symposium (Kinosaki)*, pages 1 – 72. 2000.

[SD74] Bernard Saint-Donat. Projective Models of K-3 Surfaces. *Am. Jour. Math.*, 96(4):602–639, 1974.

[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.

[SS] J. Schicho and D. Sevilla. Effective radical parametrization of trigonal curves. Preprint: online at arXiv as arXiv:1104.2470v1[math.AG].

[Sch98] Josef Schicho. Rational parametrization of surfaces. *J. Symbolic Comput.*, 26(1):1–29, 1998.

[Sch00a] Josef Schicho. Proper parametrization of surfaces with a rational pencil. In *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation (St. Andrews)*, 292–300 (electronic). New York, 2000. ACM.

[ST02] Frank-Olaf Schreyer and Fabio Tonoli. Needles in a Haystack: Special Varieties via Small Fields. In Eisenbud and others, editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of Springer Algorithms and Computation in Mathematics Series, 251–277. Springer-Verlag, 2002.

[Ser55] J-P. Serre. Faisceaux Algebriques Coherents. *Ann. Maths.*, 61:197–278, 1955.

[SvdV87] A.J. Sommese and A. van der Ven. On the adjunction mapping. *Math. Ann.*, 278:593–603, 1987.

[Sto11] "Reduction theory of point clusters in projective space."

[Suz] K. Suzuki. On Fano indices of Q-Fano 3-folds. Available as math.AG/0210309 on arXiv.

[SB] B. Szendroi and A. Buckley. Orbifold Riemann-Roch for threefolds with an application to Calabi-Yau geometry. math.AG/0309033.

[Tui16] Jan Tuitman. Counting points on curves using a map to $\mathbf{P^1}$. *Math. Comp.*, 85(298):961–981, 2016.

[Tui17] Jan Tuitman. Counting points on curves using a map to $\mathbf{P^1}$, II. *Finite Fields Appl.*, 45:301–322, 2017.

[vdE00] Arno van den Essen. *Polynomial automorphisms and the Jacobian conjecture.* Volume 190 of Progress in Mathematics. Birkhaeuser, ABirkhaeuser, 2000.

[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.

Index

Symbols

P (*operation*), 10
& join S (*operation*), 17
- D (*operation*), 104, 198, 400
- f (*operation*), 25
- p1 (*operation*), 194
- x (*operation*), 189

A

A ! 1 (*operation*), 173
A ! f (*operation*), 173
A ! [a,...] (*operation*), 148
A ! [a,b,...] (*operation*), 14
A . i (*operation*), 10, 173
A eq B (*operation*), 11
a eq b (*operation*), 189
a in A (*operation*), 174
a in S (*operation*), 189
A subset B (*operation*), 174
A(L) ! [a,...] (*operation*), 148
A(L) ! [a,b,...] (*operation*), 14
Abel-Jacobi Map of a Principal Divisor (*example*), 345
Abel-Jacobi Map of Function Field Divisors (*example*), 346
Abel-Jacobi Map on a Superelliptic Curve (*example*), 344
AbelJacobi(D, P) (*function*), 344
AbelJacobi(DF, X) (*function*), 344
AbelJacobi(DX) (*function*), 344
AbelJacobi(P) (*function*), 344
AbelJacobi(P, Q) (*function*), 344
AbsoluteRationalScroll(k,N) (*function*), 12
AbsoluteRationalScroll(k,S) (*function*), 392
Accessing the K3 Database (*example*), 368
Adjoint Spaces of a Surface (*example*), 291
AdjointIdeal(C) (*function*), 155
AdjointIdealForNodalCurve(C) (*function*), 155
AdjointLinearSystem(C) (*function*), 155
AdjointLinearSystemForNodalCurve(C, d) (*function*), 155
AdjointLinearSystemFromIdeal(I, d) (*function*), 155
Adjoints(C,d) (*function*), 155
Affine and Projective Subsystem Timings (*example*), 91
Affine Patches of the Projective Plane (*example*), 5
AffineDecomposition(f) (*function*), 73
AffineDecomposition(P) (*function*), 50
AffinePatch(C,i) (*function*), 167
AffinePatch(X,i) (*function*), 46
AffinePatch(X,p) (*function*), 47
AffinePlane(k) (*function*), 147
AffineSpace(k,n) (*function*), 9, 147
AffineSpace(R) (*function*), 10
AFRNumber(X) (*function*), 363
AlgebraMap(f) (*function*), 65
AlgorithmicFunctionField(F) (*function*), 188
AllCones(F) (*function*), 387
AllDefiningPolynomials(f) (*function*), 65
AllInverseDefiningPolynomials(f) (*function*), 65
AllRays(F) (*function*), 388
Ambient(F) (*function*), 388
Ambient(L) (*function*), 96
Ambient(X) (*function*), 21
AmbientSpace(C) (*function*), 151
AmbientSpace(L) (*function*), 96
AmbientSpace(X) (*function*), 21
ApparentCodimension(f) (*function*), 355
ApparentCodimension(X) (*function*), 361
ApparentEquationDegrees(f) (*function*), 355
ApparentEquationDegrees(X) (*function*), 361
ApparentSyzygyDegrees(f) (*function*), 355
ApparentSyzygyDegrees(X) (*function*), 361
ApplyContravariant(c, d) (*function*), 315
AreLinearlyEquivalent(D,E) (*function*), 199, 403
ArithmeticGenus(C) (*function*), 165
ArithmeticGenus(dsd,i) (*function*), 270
ArithmeticGenus(G,i) (*function*), 270
ArithmeticGenus(S) (*function*), 239
ArithmeticGenus(X) (*function*), 41
ArithmeticGenusOfDesingularization(S) (*function*), 291
Arrows(s) (*function*), 230
ArtinTateFormula(f, q, h20) (*function*), 364
AssignNames(~A,N) (*procedure*), 10
AssignNames(~C, S) (*procedure*), 397
AssignNames(~F, S) (*procedure*), 24
AssignNames(~X,N) (*procedure*), 20
Atkin-Lehner Quotients of X0(54) (*example*), 178
Aut(P) (*function*), 75
Automorphism(A, g) (*function*), 73
Automorphism(A,F) (*function*), 73
Automorphism(A,M) (*function*), 73

Automorphism(A,p) (*function*), 73
Automorphism(A,q) (*function*), 169
Automorphism(P,F) (*function*), 75
Automorphism(P,M) (*function*), 75
Automorphism(X,F) (*function*), 71
AutomorphismGroup(C) (*function*), 172
AutomorphismGroup(C,auts) (*function*), 172
AutomorphismGroup(P) (*function*), 75
Automorphisms of a Superelliptic Curve (*example*), 175
Automorphisms of Projective Space as Matrices (*example*), 75
Automorphisms of the Fermat Quartic (*example*), 175
Automorphisms(C) (*function*), 172

B

Base Change of a Singular Curve (*example*), 150
Base Points of a Map (*example*), 69
BaseChange(A,K) (*function*), 45
BaseChange(A,m) (*function*), 45
BaseChange(C, A) (*function*), 150
BaseChange(C, A, m) (*function*), 150
BaseChange(C, K) (*function*), 150
BaseChange(C, m) (*function*), 150
BaseChange(C, n) (*function*), 150
BaseChange(F,K) (*function*), 45
BaseChange(F,m) (*function*), 45
BaseChange(X, n) (*function*), 45
BaseChange(X,A) (*function*), 45
BaseChange(X,A,m) (*function*), 45
BaseComponent(L) (*function*), 96
BaseExtend(A,K) (*function*), 45
BaseExtend(A,m) (*function*), 45
BaseExtend(F,K) (*function*), 45
BaseExtend(F,m) (*function*), 45
BaseExtend(X, n) (*function*), 45
BaseExtend(X,A) (*function*), 45
BaseExtend(X,A,m) (*function*), 45
BaseField(C) (*function*), 151
BaseField(X) (*function*), 21
BaseLocus(D) (*function*), 105
BasePoint(X) (*function*), 334
BasePoints(f) (*function*), 69
BasePoints(f,L) (*function*), 69
BasePoints(L) (*function*), 98
BaseRing(C) (*function*), 151, 396
BaseRing(X) (*function*), 21
BaseScheme(f) (*function*), 68
BaseScheme(L) (*function*), 96
Basis(D) (*function*), 203
BasisOfDegree0CoxMonomials(X) (*function*), 398
BasisOfDifferentialsFirstKind(C) (*function*), 189
BasisOfHolomorphicDifferentials(C) (*function*), 189
BasisOfHolomorphicTwoForms(S) (*function*), 260
BasisOfRationalFunctionField(X) (*function*), 398
Basket(Q) (*function*), 359
Basket(Q1,Q2) (*function*), 359
Basket(X) (*function*), 360
BettiNumbers(X) (*function*), 361
BigPeriodMatrix(X) (*function*), 334
BigTorus(k,N) (*function*), 393
BigTorus(X) (*function*), 393
BinomialToricEmbedding(Z) (*function*), 413
Blow-Up Desingularisation Accessors (*example*), 256
Blowing Down Lines by a Divisor Map (*example*), 127
Blowing Down Lines on a Degenerate Del Pezzo (*example*), 281
Blowing Up a Line in Projective Space (*example*), 19
Blowup(C) (*function*), 159
Blowup(C, M) (*function*), 159
Blowup(F,v) (*function*), 384
Blowup(X,p) (*function*), 18
Blowup(X,v) (*function*), 406
Blowup(X,Y) (*function*), 18
BlowUpDivisor(S, dsd, i) (*function*), 256
BlowUpDivisorAllPatches(dsd, i) (*function*), 256
BogomolovNumber(X) (*function*), 372
Bounding Picard Rank with Fewer Point Counts (*example*), 365
BranchPoints(X) (*function*), 335

C

C ! [a,...] (*operation*), 157
C . i (*operation*), 397
C eq D (*operation*), 166, 359
C in F (*operation*), 386
C(L) ! [a,...] (*operation*), 157
C1 eq C2 (*operation*), 396
CalabiYau(p1,p2,B) (*function*), 372
CalculateCanonicalClass(~g) (*procedure*), 227
CalculateMultiplicities(~g) (*procedure*), 227
CalculateTransverseIntersections(~g) (*procedure*), 228
Canonical Divisor (*example*), 200
Canonical Divisor as a Hyperplane Section (*example*), 201
Canonical Forms and Plurigenera of a Del Pezzo (*example*), 261
Canonical Image of an Ordinary Plane Curve (*example*), 155
Canonical Map (*example*), 204
CanonicalClass(g) (*function*), 228
CanonicalClass(X) (*function*), 400
CanonicalCoordinateIdeal(S) (*function*), 249
CanonicalDissidentPoints(C) (*function*), 360
CanonicalDivisor(C) (*function*), 198
CanonicalDivisor(X) (*function*), 101, 400

CanonicalImage(C, eqns) (*function*), 204
CanonicalImage(C, phi) (*function*), 204
CanonicalIntersection(dsd,i) (*function*), 263, 270
CanonicalIntersection(G,i) (*function*), 270
CanonicalIntersection(S,D) (*function*), 263
Canonicalisation(F) (*function*), 390
Canonicalisation(X) (*function*), 407
CanonicalLinearSystem(C) (*function*), 155
CanonicalLinearSystemFromIdeal(I, d) (*function*), 155
CanonicalMap(C) (*function*), 204
CanonicalMap(C,P) (*function*), 204
CanonicalSheaf(X) (*function*), 119
CanonicalSheaf(X, n) (*function*), 119
CanonicalWeightedModel(S) (*function*), 249
CartanMatrix(g) (*function*), 228
Cartier Divisors (*example*), 105
Cartier(a) (*function*), 190
Cartier(a, r) (*function*), 190
Cartier(D) (*function*), 401
CartierRepresentation(C) (*function*), 190
CartierRepresentation(C, r) (*function*), 190
CartierToWeilMap(X) (*function*), 402
CentredAffinePatch(S, p) (*function*), 50
CheckCodimension(X) (*function*), 362
CheckWeilPolynomial(f, q, h20) (*function*), 364
ChernNumber(S,n) (*function*), 240
ClassGroup(C) (*function*), 201
ClassGroupAbelianInvariants(C) (*function*), 202
ClassGroupPRank(C) (*function*), 202
ClassicalCovariantsOfCubicSurface(f) (*function*), 314
Classifying Rational Surfaces (*example*), 293
ClassifyRationalSurface(S) (*function*), 293
ClassNumber(C) (*function*), 201
ClebschSalmonInvariants(f) (*function*), 313
CliffordIndexOne(C) (*function*), 214
CliffordIndexOne(C,X) (*function*), 214
Closure of a Degenerate Patch (*example*), 48
Cluster(D) (*function*), 200
Cluster(p) (*function*), 29
Cluster(S) (*function*), 29
Cluster(X, p) (*function*), 29
Cluster(X, S) (*function*), 29
Cluster(X,F) (*function*), 16
Cluster(X,f) (*function*), 16
Cluster(X,I) (*function*), 16
Cluster(X,Q) (*function*), 16
CoblesRadicand(p) (*function*), 312
Code from the Klein Quartic (*example*), 208
Codimension(X) (*function*), 41, 361
Codomain(f) (*function*), 64, 124
CoefficientField(X) (*function*), 21
CoefficientMap(L) (*function*), 98
CoefficientRing(C) (*function*), 151, 396
CoefficientRing(X) (*function*), 21
CoefficientSpace(L) (*function*), 98
CohomologyDimension(S, r, n) (*function*), 130
Cokernel(f) (*function*), 124
CollinearPointsOnPlane(P2,S) (*function*), 302
CombineIdealFactorisation(~D) (*procedure*), 102
CommonComponent(X,Y) (*function*), 22
CommonZeros(C, L) (*function*), 194
CommonZeros(L) (*function*), 194
Complement(L,K) (*function*), 98
Complement(L,X) (*function*), 98
Complement(X, Y) (*function*), 18
ComplementaryDivisor(D,P) (*function*), 200
ComplementaryDivisor(D,p) (*function*), 200
Completion(F, p) (*function*), 182
Component(v) (*function*), 227
ComponentGroup(M) (*function*), 212
Components(f) (*function*), 61
ComputePrimeFactorisation(~D) (*procedure*), 102
ComputeReducedFactorisation(~D) (*procedure*), 102
Cone of a Terminal Quotient Singularity (*example*), 391
Cone(F,i) (*function*), 387
Cone(F,S) (*function*), 387
ConeIndices(F) (*function*), 387
ConeIndices(F,C) (*function*), 387
ConeIntersection(F,C1,C2) (*function*), 387
Cones(F) (*function*), 386
Cones(F,i) (*function*), 386
ConesOfCodimension(F,i) (*function*), 386
Conic(P,S) (*function*), 149
Connect(v,w) (*function*), 227
ConstantMap(X,Y,p) (*function*), 58
Constructing the Drell-Yan K3 Surface (*example*), 317
ContravariantsOfCubicSurface(f) (*function*), 314
Converting a Map to a Morphism (*example*), 62
Converting Function Field Elements (*example*), 25
Coordinate(p, i) (*function*), 157
Coordinate(p,i) (*function*), 15
CoordinateRing(A) (*function*), 13, 147
CoordinateRing(C) (*function*), 151
CoordinateRing(X) (*function*), 22
Coordinates(P) (*function*), 341
Coordinates(p) (*function*), 15, 148, 157
Corank2Case(f) (*function*), 38
Corank3Case(f) (*function*), 39
Counting Points on a Cubic Surface (*example*), 311
Cox Ring of a Weighted Projective Plane (*example*), 395
Cox's Quotient Construction (*example*), 380
CoxMonomialLattice(C) (*function*), 398

CoxMonomialLattice(X) (*function*), 393
CoxRing(k,F) (*function*), 395
CoxRing(R,B,Z,Q) (*function*), 396
CoxRing(X) (*function*), 395
CreateK3Data(g) (*function*), 373
CreateK3Data(g,B) (*function*), 373
CreateK3Data(g,r) (*function*), 373
CreateVirtualRays(S) (*function*), 388
Creating a Function Field (*example*), 182
Creating a Point on a Riemann Surface (*example*), 340
Creating an Affine Space (*example*), 2
Creating Divisors on a Curve (*example*), 197
Creating Maps of Affine Spaces (*example*), 57
Creating Points by Coercion (*example*), 4
Creating Schemes (*example*), 17
Creating Surfaces (*example*), 237
Cubic Surface from Hexahedral Coefficients (*example*), 312
CubicSurfaceByHexahedralCoefficients(p) (*function*), 312
CubicSurfaceFromClebschSalmon(inv) (*function*), 313
Curve Singularity with Transverse Type (*example*), 358
Curve with a Cusp Singularity (*example*), 93
Curve(A) (*function*), 174
Curve(a) (*function*), 189
Curve(A,f) (*function*), 149
Curve(A,I) (*function*), 149
Curve(D) (*function*), 196
Curve(d,p,m) (*function*), 358
Curve(d,p,N) (*function*), 358
Curve(d,p,N,t) (*function*), 358
Curve(Div) (*function*), 196
Curve(F) (*function*), 182
Curve(P) (*function*), 27, 157, 192, 194
Curve(p) (*function*), 27, 157
Curve(S) (*function*), 189
Curve(X) (*function*), 22, 149
Curve(X,S) (*function*), 149
CurveDifferential(C, d) (*function*), 188
CurveDivisor(C, d) (*function*), 188
CurvePlace(C, p) (*function*), 188
CurveQuotient(G) (*function*), 178
CurveQuotient(L) (*function*), 178
Curves in Space (*example*), 116
Curves(B) (*function*), 359

D

D ! 0 (*operation*), 196
D * v (*operation*), 400
D + E (*operation*), 198
D - E (*operation*), 198
D div n (*operation*), 198
D eq E (*operation*), 199
D ge E (*operation*), 199
D gt E (*operation*), 199
d in D (*operation*), 199
D le E (*operation*), 199
D lt E (*operation*), 199
D mod n (*operation*), 198
D ne E (*operation*), 199
d notin D (*operation*), 199
D1 + D2 (*operation*), 104, 400
D1 - D2 (*operation*), 104, 400
D1 eq D2 (*operation*), 104
Danilov's Nonprojective Toric Variety (*example*), 378
Decomposing an Affine Linear Map (*example*), 73
Decomposition of a Divisor (*example*), 199
Decomposition(D) (*function*), 197
Defining the Twisted Cubic (*example*), 3
DefiningEquation(X) (*function*), 22
DefiningEquations(f) (*function*), 64
DefiningEquations(X) (*function*), 22
DefiningIdeal(C) (*function*), 151
DefiningIdeal(X) (*function*), 22
DefiningMonomial(D) (*function*), 403
DefiningPolynomial(C) (*function*), 151
DefiningPolynomial(X) (*function*), 22
DefiningPolynomials(f) (*function*), 64
DefiningPolynomials(X) (*function*), 22
Degree 20 Curve with 18 Singular Points (*example*), 89
Degree(C) (*function*), 151, 358
Degree(D) (*function*), 105, 199, 343
Degree(D,H) (*function*), 105
Degree(f) (*function*), 124, 182
Degree(L) (*function*), 96
Degree(m) (*function*), 170
Degree(P) (*function*), 195
Degree(s) (*function*), 230
Degree(X) (*function*), 41, 334, 360
Degree(Z) (*function*), 31
Degree6DelPezzoType2_1(K,pt) (*function*), 305
Degree6DelPezzoType2_2(K,pt) (*function*), 305
Degree6DelPezzoType2_3(K,pt) (*function*), 305
Degree6DelPezzoType3(K,pt) (*function*), 305
Degree6DelPezzoType4(K,K1,pt) (*function*), 305
Degree6DelPezzoType6(K,pt) (*function*), 305
Degrees(X,f) (*function*), 14
DegreeTwoK3Surface(f) (*function*), 316
DelPezzoSurface(f) (*function*), 301
DelPezzoSurface(P,L) (*function*), 301
DelPezzoSurface(S) (*function*), 301
DelPezzoSurface(Z) (*function*), 301
DeltaAdjustment(C, p) (*function*), 159
Denominator(D) (*function*), 199
Denominator(f) (*function*), 24
Denominator(f, X) (*function*), 25
DesingulariseSurfaceByBlowUp(S) (*function*), 255
Desingularising a Degree 9 Elliptic Surface (*example*), 272

Determinant(g) (*function*), 228
Difference of Schemes (*example*), 18
Difference(X, Y) (*function*), 18
Differential(a) (*function*), 189
DifferentialBasis(D) (*function*), 189, 203
DifferentialMultiplicities(dsd) (*function*), 260
DifferentialSpace(C) (*function*), 189
DifferentialSpace(D) (*function*), 189, 203
Dimension(C) (*function*), 359
Dimension(D) (*function*), 203
Dimension(L) (*function*), 96
Dimension(p) (*function*), 357
Dimension(X) (*function*), 41, 361
DimensionOfFieldOfGeometricIrreducibility(C) (*function*), 183
DimensionOfGlobalSections(S) (*function*), 130
DirectProduct(A,B) (*function*), 12, 147
DirectSum(S, T) (*function*), 123
Disconnect(v,w) (*function*), 227
Discrete Logarithm by Index Calculus (*example*), 206
DiscriminantPoints(f) (*function*), 335
DiscriminantPoints(f, sigma) (*function*), 335
DiscriminantPoints(X) (*function*), 335
Distinguishing Nodes and Cusps (*example*), 159
Div ! p (*operation*), 196
Div1 eq Div2 (*operation*), 196
Div1 ne Div2 (*operation*), 196
Divisor Class Group over a Finite Field (*example*), 201
Divisor Map of a Rational Scroll (*example*), 126
Divisor Multiplicities on a Cubic Surface (*example*), 103
Divisor(a) (*function*), 197
Divisor(C, f) (*function*), 197
Divisor(C, I) (*function*), 197
Divisor(C, p, q) (*function*), 197
Divisor(C, S) (*function*), 196
Divisor(C, X) (*function*), 197
Divisor(d) (*function*), 190
Divisor(D, f) (*function*), 197
Divisor(D, I) (*function*), 197
Divisor(D, p, q) (*function*), 197
Divisor(D, S) (*function*), 196
Divisor(D, X) (*function*), 197
Divisor(f) (*function*), 197
Divisor(G,i) (*function*), 399
Divisor(G,S) (*function*), 399
Divisor(p) (*function*), 196
Divisor(S) (*function*), 196
Divisor(S,V) (*function*), 343
Divisor(X,f) (*function*), 100, 399
Divisor(X,I) (*function*), 100
Divisor(X,i) (*function*), 399
Divisor(X,m) (*function*), 399
Divisor(X,Q) (*function*), 100
Divisor(X,S) (*function*), 399
Divisor(X,Y) (*function*), 100
DivisorClassGroup(C) (*function*), 398
DivisorClassLattice(C) (*function*), 398
DivisorClassLattice(X) (*function*), 393
DivisorGroup(C) (*function*), 195
DivisorGroup(D) (*function*), 196
DivisorGroup(X) (*function*), 100, 399
DivisorMap(D) (*function*), 203
DivisorMap(D,P) (*function*), 203
DivisorMap(S) (*function*), 125
DivisorOfDegreeOne(C) (*function*), 184
DivisorToSheaf(X, I) (*function*), 125
Domain(f) (*function*), 64, 124
Dual of the Cayley Cubic (*example*), 314
Dual Resolution Graph of a Degree 9 Surface (*example*), 266
Dual(S) (*function*), 122
DualFaceInDualFan(P,Q) (*function*), 388
DualFan(P) (*function*), 383
DualResolutionGraph(dsd) (*function*), 265

E

EckardtPoints(X) (*function*), 302
EdgeDeterminant(u,v) (*function*), 230
EdgeLabels(s) (*function*), 230
EffectiveHypersurfaceTwist(D) (*function*), 101
EffectiveSubcanonicalCurves(g) (*function*), 362
EffectiveSubcanonicalCurves(g,d) (*function*), 362
Eigenspace(p) (*function*), 357
Elementary Transformation of Scrolls (*example*), 69
Elements of a Function Field (*example*), 183
Elimination(X,V) (*function*), 59
Elliptic Fibrations of the Drell-Yan Surface (*example*), 323
EllipticCurve(C) (*function*), 168
EllipticCurve(C,p) (*function*), 168
EllipticFibrationRRSpaceDeg2K3(S,divlst,exdivlst) (*function*), 325
EllipticGeneralFibreDeg2K3(S,B,secs) (*function*), 325
Embedded Resolution of a Projective Curve (*example*), 286
Embedded Resolution of an Affine Curve (*example*), 285
Embedding a Plane Curve in Projective 3-Space (*example*), 84
Embedding(X) (*function*), 334
EmbedPlaneCurveInP3(C) (*function*), 84
Emptiness of p-adic Point Sets (*example*), 52
Empty p-adic Points of a Hyperelliptic Curve (*example*), 52
EmptyBasket() (*function*), 359
EmptyScheme(X) (*function*), 17

EmptySubscheme(X) (*function*), 17
Equality of Linear Systems (*example*), 86
Equality of Points (*example*), 14
Equation(X) (*function*), 22
Equations(X) (*function*), 22
EulerCharacteristic(s) (*function*), 230
Evaluate(a, P) (*function*), 195
Evaluate(f, p) (*function*), 15, 182
EvaluateByPowerSeries(m, P) (*function*), 170
Evaluating a Map by Power Series (*example*), 170
Evaluating Functions at Points (*example*), 15
Exceptional Curves on a K3 Surface (*example*), 264
ExceptionalDivisors(S) (*function*), 263
Expand(f, p) (*function*), 182
Expand(hms) (*function*), 124
Expand(phi) (*function*), 61
Extend(phi) (*function*), 61
Extremal Contractions of a Blowup (*example*), 408
ExtremalRayContraction(X,i) (*function*), 407
ExtremalRayContractionDivisor(X,i) (*function*), 407
ExtremalRayContractions(X) (*function*), 407
ExtremalRays(X) (*function*), 407

F

F ! g (*operation*), 24
F ! r (*operation*), 182
f * g (*operation*), 25, 173
f * x (*operation*), 189
f + g (*operation*), 25
F . i (*operation*), 24
f / g (*operation*), 25
F @ f (*operation*), 389
f - g (*operation*), 25
F ^ n (*operation*), 386
f ^ n (*operation*), 25, 173
F eq G (*operation*), 386
f eq g (*operation*), 25, 65
f in L (*operation*), 99
f(K) (*function*), 66
f(m) (*function*), 66
f(p) (*function*), 15, 65, 182
f(X) (*function*), 67, 174
F1 * F2 (*operation*), 386
Face(F,C) (*function*), 388
FactoredDefiningPolynomials(f) (*function*), 64
FactoredInverseDefiningPolynomials(f) (*function*), 65
FactoredOrder(A) (*function*), 174
Factorising a Cremona Transformation (*example*), 78
FakeProjectiveSpace(k,W,Q) (*function*), 392
Fan from Weight Data (*example*), 384
Fan(C) (*function*), 383, 398
Fan(F1,F2) (*function*), 386
Fan(Q) (*function*), 383, 386
Fan(R,S) (*function*), 383
Fan(X) (*function*), 393
Fano 3-Folds with the Same Basket (*example*), 371
Fano Index of a Toric Fano 3-Fold (*example*), 379
Fano(D,f,i) (*function*), 372
Fano(D,f,Q,i) (*function*), 372
Fano(D,i) (*function*), 372
Fano(f,B) (*function*), 371
Fano(f,B,g) (*function*), 371
FanoBaseGenus(X) (*function*), 372
FanoDatabase() (*function*), 372
FanOfAffineSpace(n) (*function*), 383
FanOfFakeProjectiveSpace(W,Q) (*function*), 383
FanOfProjectiveSpace(n) (*function*), 383
FanOfWPS(W) (*function*), 383
FanoGenus(X) (*function*), 371
FanoIndex(X) (*function*), 371
FanWithWeights(W) (*function*), 384
FieldOfGeometricIrreducibility(C) (*function*), 183
FindFirstGenerators(g) (*function*), 355
Finding Images with Linear Systems (*example*), 86
FindN(p1,p2,B) (*function*), 372
FindN(X) (*function*), 372
First Generators from a Hilbert Series (*example*), 355
FirstChernClassOfDesingularization(S) (*function*), 263
FirstWeights(X) (*function*), 362
Flexes(C) (*function*), 166
Flip(D) (*function*), 408
Flip(X,i) (*function*), 408
FlipCoordinates(A) (*function*), 169
Formal Desingularization of a Projective Surface (*example*), 288
Formal Desingularization of an Affine Surface (*example*), 287
FormallyResolveProjectiveHyperSurface(S) (*function*), 287
FormalPoint(P) (*function*), 158
FractionalPart(D) (*function*), 101
Frobenius Map (*example*), 58
FrobeniusTracesToWeilPolynomials(tr, q, i, deg) (*function*), 364
FullModule(S) (*function*), 121
FunctionDegree(f) (*function*), 65
FunctionField(A) (*function*), 13, 148
FunctionField(C) (*function*), 181
FunctionField(X) (*function*), 334
FunctionFieldDifferential(d) (*function*), 188
FunctionFieldDivisor(d) (*function*), 188
FunctionFieldPlace(p) (*function*), 188
Fundamental Group of a Riemann Surface (*example*), 336

FundamentalGroup(P) (*function*), 335
FundamentalGroup(X) (*function*), 335

G

g ! i (*operation*), 223
g * f (*operation*), 60
g eq h (*operation*), 173, 223
g ne h (*operation*), 173
G1 eq G2 (*operation*), 100, 399
GapNumbers(C) (*function*), 184
GapNumbers(C, P) (*function*), 195
GapNumbers(D) (*function*), 203
GapNumbers(D,p) (*function*), 203
GapNumbers(P) (*function*), 195
GapNumbers(p) (*function*), 203
GCD(D1, D2) (*function*), 200
Gcd(D1, D2) (*function*), 200
Generators(A) (*function*), 174
GenericPoint(X) (*function*), 26
Genus of a Cuspidal Cubic (*example*), 166
Genus(C) (*function*), 165, 183
Genus(dsd,i) (*function*), 270
Genus(G,i) (*function*), 270
Genus(X) (*function*), 334, 363
Genus2GonalMap(C) (*function*), 215
Genus3GonalMap(C) (*function*), 215
Genus4GonalMap(C) (*function*), 216
Genus5GonalMap(C) (*function*), 216
Genus5PlaneCurveModel(C) (*function*), 219
Genus5PlaneCurveModel(C,P) (*function*), 219
Genus5PlaneCurveModel(C,Z) (*function*), 219
Genus6GonalMap(C) (*function*), 216
Genus6PlaneCurveModel(C) (*function*), 218
GenusAndCanonicalMap(C) (*function*), 214
GenusContribution(g) (*function*), 228
GenusViaArithmeticGenus(C) (*function*), 165
GeometricGenus(C) (*function*), 165
GeometricGenus(S) (*function*), 239
GeometricGenusOfDesingularization(S) (*function*), 290
GlobalSectionSubmodule(S) (*function*), 121
GlobalUnitGroup(C) (*function*), 201
Gonal Maps of Genus 5 and 6 Curves (*example*), 217
GonalityPreservingLift(C) (*function*), 186
GorensteinIndex(C) (*function*), 390
GorensteinIndex(P) (*function*), 390
GradedCone(D) (*function*), 403
Gradings(C) (*function*), 397
Gradings(X) (*function*), 13, 394
Graph Map of an Elliptic Curve (*example*), 80
GreatestCommonDivisor(D1, D2) (*function*), 200
GroebnerBasis(X) (*function*), 22

H

HasAffinePatch(X, i) (*function*), 47
HasAllRootsOnUnitCircle(f) (*function*), 364
HasFunctionField(A) (*function*), 13
HasFunctionField(C) (*function*), 182
HasGCD(X) (*function*), 21
HasGroebnerBasis(X) (*function*), 21
HasIrregularFibres(s) (*function*), 230
HasKnownInverse(f) (*function*), 60
HasNonsingularPoint(X) (*function*), 28
HasNonsingularPoint(X,L) (*function*), 28
HasOnlyOrdinarySingularities(C) (*function*), 154
HasOnlyOrdinarySingularitiesMonteCarlo(C) (*function*), 154
HasOnlySimpleSingularities(S) (*function*), 242
HasPlace(C, m) (*function*), 192
HasPointsOverExtension(X) (*function*), 30
HasPointsOverExtension(X,L) (*function*), 30
HasResultant(X) (*function*), 21
HasseWittInvariant(C) (*function*), 202
HasSingularPointsOverExtension(C) (*function*), 166
HeightOnAmbient(P) (*function*), 50
Hessian of a Plane Curve (*example*), 152
HessianMatrix(C) (*function*), 152
HessianMatrix(X) (*function*), 23
Hilbert Function of a Genus 4 Curve (*example*), 353
HilbertCoefficient(D,i) (*function*), 404
HilbertCoefficients(D,l) (*function*), 404
HilbertDeltaVector(D) (*function*), 404
HilbertFunction(p,V) (*function*), 352
HilbertFunction(Q,V) (*function*), 352
HilbertNumerator(g, D) (*function*), 353
HilbertNumerator(X) (*function*), 361
HilbertPolynomial(D) (*function*), 404
HilbertPolynomialOfCurve(g,m) (*function*), 362
HilbertSeries(D) (*function*), 404
HilbertSeries(p,V) (*function*), 353
HilbertSeries(Q,V) (*function*), 353
HilbertSeries(X) (*function*), 361
HilbertSeriesMultipliedByMinimal Denominator(p,V) (*function*), 353
HilbertSeriesMultipliedByMinimal Denominator(Q,V) (*function*), 353
Hirzebruch Surface as a GIT Quotient (*example*), 392
HirzebruchSurface(k,n) (*function*), 393
HodgeNumber(S,i,j) (*function*), 240
Holomorphic Differentials of a Plane Curve (*example*), 337
Holomorphic Differentials of a Plane Quartic (*example*), 190
Holomorphic Differentials of a Superelliptic Surface (*example*), 337
HolomorphicDifferentials(X) (*function*), 337
Hom Spaces of Schemes (*example*), 61
HomAdjoints(m,n,S) (*function*), 290
Homology Basis for the Big Period Matrix (*example*), 339
Homology Basis from Monodromy (*example*), 338

HomologyBasis(L) (*function*), 338
HomologyBasis(X) (*function*), 338
HorrocksMumfordBundle(P) (*function*), 121
Hyperelliptic Involution as an Automorphism (*example*), 71
HyperplaneAtInfinity(X) (*function*), 50
HyperplaneSectionDivisor(X) (*function*), 101
Hypersurface Singularities of a Del Pezzo (*example*), 34
HypersurfaceSingularityExpand Function(dat,f,prec,R) (*function*), 33
HypersurfaceSingularityExpand Further(dat,prec,R) (*function*), 33

I

Id(A) (*function*), 173
Id(D) (*function*), 196
Ideal(D) (*function*), 101, 200
Ideal(P) (*function*), 193
Ideal(X) (*function*), 22
IdealFactorisation(D) (*function*), 102
IdealOfSupport(D) (*function*), 102
IdenticalAmbientSpace(X,Y) (*function*), 21
Identity(A) (*function*), 173
Identity(D) (*function*), 196
Identity(S) (*function*), 189
IdentityAutomorphism(A) (*function*), 169
IdentityAutomorphism(X) (*function*), 71
IdentityMap(X) (*function*), 58, 71
IharaBound(C) (*function*), 184
IharaBound(C, m) (*function*), 184
Image Equations of a Canonical Curve (*example*), 68
Image of a Point (*example*), 66
Image(f) (*function*), 67, 124
Image(f,X,d) (*function*), 67
ImageFan(D) (*function*), 402
ImageSystem(f,S,d) (*function*), 86
ImproveParametrization(p) (*function*), 63
Improving a Conic Parametrization (*example*), 64
IncludeWeight(~X,w) (*procedure*), 362
IncludeWeight(X,w) (*function*), 363
IndeterminacyLocus(f) (*function*), 406
Index(C) (*function*), 358
Index(D,X) (*function*), 369
Index(p) (*function*), 357
Index(v) (*function*), 224, 229
IndexCalculus(D1, D2, D0, np) (*function*), 205
IndexCalculus(D1, D2, D0, np, n, rr) (*function*), 205
IndexCalculusMatrix(D1, D2, D0, n, rr) (*function*), 205
IndexOfSpeciality(D) (*function*), 203
IneffectiveDivisorToSheaf(X, I, J) (*function*), 126
IneffectiveRiemannRochBasis(X, I, J) (*function*), 126
IneffectiveSubcanonicalCurves(g) (*function*), 362
IneffectiveSubcanonicalCurves(g,d) (*function*), 362
InflectionPoints(C) (*function*), 166
InitialCoefficients(X) (*function*), 361
InnerNormals(F) (*function*), 388
IntegerRing(F) (*function*), 24
Integers(F) (*function*), 24
IntegralMultiple(D) (*function*), 101
IntegralSplit(f, X) (*function*), 25
Intersection Matrix of the Drell-Yan Surface (*example*), 319
Intersection Numbers from Hilmar and Smyth (*example*), 164
Intersection of Two Plane Curves (*example*), 31
Intersection Points after Base Change (*example*), 45
Intersection(L,K) (*function*), 99
Intersection(X,Y) (*function*), 17
IntersectionForm(X,C) (*function*), 402
IntersectionForms(X) (*function*), 402
IntersectionMatrix(dsd) (*function*), 258
IntersectionMatrix(M) (*function*), 211
IntersectionMatrixOnDegree2K3(S, Ds) (*function*), 318
IntersectionNumber(C,D,p) (*function*), 163
IntersectionNumber(D1,D2) (*function*), 105
IntersectionNumberOfStrictTransforms(S,D1, D2) (*function*), 259
IntersectionNumbers(C,D) (*function*), 163
IntersectionNumbers(F,G) (*function*), 163
IntersectionPairing(S, T) (*function*), 130
Invariants of a Cubic Surface (*example*), 313
Invariants of a Kummer Surface (*example*), 240
Invariants of a Riemann Surface (*example*), 334
Inverse(f) (*function*), 60, 173
InverseDefiningPolynomials(f) (*function*), 64
Inverting a Map to an Elliptic Curve (*example*), 60
IrreducibleComponents(X) (*function*), 43
Irregularity(S) (*function*), 239
IrrelevantComponents(C) (*function*), 396
IrrelevantGenerators(C) (*function*), 397
IrrelevantIdeal(C) (*function*), 396
IrrelevantIdeal(X) (*function*), 393
IsAbsolutelyIrreducible(C) (*function*), 183
IsAffine(X) (*function*), 20, 21
IsAffineLinear(f) (*function*), 65
IsAmbient(X) (*function*), 20
IsAmple(D) (*function*), 401
IsAnalyticallyIrreducible(C, p) (*function*), 159
IsAnalyticallyIrreducible(p) (*function*), 159
IsAnticanonical(D) (*function*), 104

IsArithmeticallyCohenMacaulay(S) (*function*), 129
IsArithmeticallyCohenMacaulay(X) (*function*), 43
IsArithmeticallyGorenstein(X) (*function*), 43
IsAutomorphism(f) (*function*), 71
IsBasePointFree(D) (*function*), 105
IsBasePointFree(L) (*function*), 96
IsBig(D) (*function*), 401
IsBogomolovUnstable(X) (*function*), 372
IsCanonical(B) (*function*), 360
IsCanonical(C) (*function*), 359, 390
IsCanonical(D) (*function*), 104, 200
IsCanonical(F) (*function*), 389
IsCanonical(p) (*function*), 357
IsCanonical(X) (*function*), 394
IsCanonicalWithTwist(D) (*function*), 104
IsCartier(D) (*function*), 105, 401
IsCluster(X) (*function*), 20
IsCoercible(X, S) (*function*), 340
IsCohenMacaulay(X) (*function*), 43
IsComplete(F) (*function*), 388
IsComplete(L) (*function*), 96
IsComplete(X) (*function*), 394
IsConic(X) (*function*), 20
IsCurve(X) (*function*), 20, 149
IsCusp(C, p) (*function*), 159
IsCusp(p) (*function*), 159
IsDelPezzo(Y) (*function*), 301
IsDivisible(D) (*function*), 104
IsDivisorialContraction(X,i) (*function*), 407
IsDominant(f) (*function*), 65
IsDoublePoint(C, p) (*function*), 158
IsDoublePoint(p) (*function*), 158
IsEffective(C) (*function*), 362
IsEffective(D) (*function*), 104, 199, 401
IsEllipticWeierstrass(C) (*function*), 168
IsEmpty(X) (*function*), 41
IsEmpty(Xm) (*function*), 51
IsEndomorphism(f) (*function*), 71
IsExact(a) (*function*), 189
IsFactorisationPrime(D) (*function*), 104
IsFakeWeightedProjectiveSpace(X) (*function*), 394
IsFanMap(F1,F2) (*function*), 389
IsFanMap(F1,F2,f) (*function*), 389
IsFano(P) (*function*), 391
IsFano(X) (*function*), 394
IsFlex(C, p) (*function*), 158
IsFlex(p) (*function*), 158
IsFlipping(X,i) (*function*), 408
IsFree(L) (*function*), 96
IsGeometricallyHyperelliptic(C) (*function*), 168
IsGorenstein(C) (*function*), 390
IsGorenstein(F) (*function*), 389
IsGorenstein(X) (*function*), 43, 394
IsGorensteinSurface(B) (*function*), 359
IsGorensteinSurface(p) (*function*), 357
IsHomogeneous(X,f) (*function*), 14
IsHyperelliptic(C) (*function*), 168
IsHyperellipticCurve(X) (*function*), 20
IsHyperellipticWeierstrass(C) (*function*), 168
IsHypersurface(X) (*function*), 22
IsHypersurfaceDivisor(D) (*function*), 201
IsHypersurfaceSingularity(p,prec) (*function*), 32
IsInflectionPoint(C, p) (*function*), 158
IsInflectionPoint(p) (*function*), 158
IsInSecantVariety(X,P) (*function*), 83
IsInSupport(v,F) (*function*), 386
IsInTangentVariety(X,P) (*function*), 81
IsIntegral(D) (*function*), 103
IsIntersection(C,D,p) (*function*), 163
IsInvertible(f) (*function*), 60, 80
IsIrreducible(C) (*function*), 152
IsIrreducible(X) (*function*), 43
IsIsolated(B) (*function*), 359
IsIsolated(C) (*function*), 390
IsIsolated(F) (*function*), 388
IsIsolated(p) (*function*), 357
IsIsolated(X) (*function*), 394
IsIsomorphic(C, D) (*function*), 172
IsIsomorphic(S, T) (*function*), 129
IsIsomorphicCubicSurface(f,g) (*function*), 311
IsIsomorphicWithTwist(S, T) (*function*), 129
IsIsomorphism(f) (*function*), 65
IsLinear(f) (*function*), 65
IsLinear(X) (*function*), 23
IsLinearlyEquivalent(D,E) (*function*), 105, 403
IsLinearlyEquivalent(D1,D2) (*function*), 201
IsLinearlyEquivalentToCartier(D) (*function*), 403
IsLinearSystemNonEmpty(D) (*function*), 107
IsLocallyFree(S) (*function*), 128
IsLocallySolvable(X, p) (*function*), 52
IsLocallySolvable(X,pl) (*function*), 53
IsMinusOne(f) (*function*), 25
IsMobile(D) (*function*), 105
IsModularCurve(X) (*function*), 21
IsMoriFibreSpace(X,i) (*function*), 408
IsNef(D) (*function*), 106, 401
IsNefAndBig(D) (*function*), 106
IsNodalCurve(C) (*function*), 153
IsNode(C, p) (*function*), 159
IsNode(p) (*function*), 159
IsNonsingular(C) (*function*), 152, 390
IsNonsingular(C, p) (*function*), 158
IsNonsingular(F) (*function*), 388
IsNonsingular(p) (*function*), 32, 158
IsNonsingular(X) (*function*), 41, 394
IsNonsingular(X,p) (*function*), 32
IsNormal(S) (*function*), 241
iso< X -> Y | F, G > (*constructor*), 57
Isolated Points of a Random Scheme (*example*), 111

IsolatedPointsFinder(S,P) (*function*), 108
IsolatedPointsLifter(S,P) (*function*), 108
IsolatedPointsLiftToMinimalPolynomials(S, P) (*function*), 108
Isomorphic Models of the Curve X(7) (*example*), 176
IsomorphicProjectionToSubspace(X) (*function*), 83
Isomorphism Testing of Cubic Surfaces (*example*), 311
Isomorphisms(C, D) (*function*), 172
IsOne(f) (*function*), 25
IsOrdinaryProjective(X) (*function*), 21
IsOrdinaryProjectiveSpace(X) (*function*), 20
IsOrdinarySingularity(C, p) (*function*), 159
IsOrdinarySingularity(p) (*function*), 32, 159
IsOrdinarySingularity(X,p) (*function*), 32
IsPlanar(X) (*function*), 22
IsPlaneCurve(X) (*function*), 20
IsPolynomial(f) (*function*), 65
IsPositive(D) (*function*), 199
IsPrime(D) (*function*), 104
IsPrincipal(D) (*function*), 104, 201, 403
IsProjective(X) (*function*), 20, 21, 394
IsQCartier(D) (*function*), 401
IsQFactorial(C) (*function*), 390
IsQFactorial(F) (*function*), 388
IsQFactorial(X) (*function*), 394
IsQGorenstein(C) (*function*), 390
IsQGorenstein(F) (*function*), 389
IsQGorenstein(X) (*function*), 394
IsQPrincipal(D) (*function*), 403
IsRational(X) (*function*), 292
IsRationalCurve(X) (*function*), 20
IsReduced(C) (*function*), 152
IsReduced(s) (*function*), 230
IsReduced(X) (*function*), 43
IsReflexive(P) (*function*), 390
IsRegular(f) (*function*), 65, 406
IsRegular(s) (*function*), 230
IsSaturated(X) (*function*), 22
IsSimpleSurfaceSingularity(p) (*function*), 241
IsSimplicial(P) (*function*), 390
IsSingular(C) (*function*), 152, 390
IsSingular(C, p) (*function*), 158
IsSingular(F) (*function*), 388
IsSingular(p) (*function*), 32, 158
IsSingular(X) (*function*), 42, 394
IsSingular(X,p) (*function*), 32
IsSmooth(P) (*function*), 390
IsSpecial(D) (*function*), 203
IsStandardAffinePatch(A) (*function*), 47
IsSubcanonicalCurve(g,d,Q) (*function*), 362
IsSubscheme(C,D) (*function*), 166
IsSubscheme(X, Y) (*function*), 23
IsSubsystem(L,K) (*function*), 99
IsTangent(C, D, p) (*function*), 158
IsTerminal(C) (*function*), 390
IsTerminal(F) (*function*), 389
IsTerminal(X) (*function*), 394
IsTerminalThreefold(B) (*function*), 359
IsTerminalThreefold(p) (*function*), 357
IsTransverse(C,D,p) (*function*), 163
IsUnit(f) (*function*), 25
IsVertex(g,v) (*function*), 224
IsWeakFano(X) (*function*), 394
IsWeierstrassPlace(D, P) (*function*), 195
IsWeierstrassPlace(P) (*function*), 195
IsWeightedProjectiveSpace(X) (*function*), 394
IsWeil(D) (*function*), 401
IsZero(a) (*function*), 190
IsZero(D) (*function*), 199
IsZero(f) (*function*), 25
IsZeroDivisor(D) (*function*), 103

J

JacobianIdeal(C) (*function*), 151
JacobianIdeal(X) (*function*), 23
JacobianMatrix(C) (*function*), 152
JacobianMatrix(X) (*function*), 23
JacobianSubrankScheme(X) (*function*), 42

K

k * p (*operation*), 194
K subset L (*operation*), 99
K3 Double Covers with Tritangent Lines (*example*), 276
K3Copy(X) (*function*), 363
K3Database() (*function*), 369
K3Surface(D,g,B) (*function*), 370
K3Surface(D,g,i) (*function*), 370
K3Surface(D,g1,g2,i) (*function*), 370
K3Surface(D,i) (*function*), 370
K3Surface(D,Q,i) (*function*), 370
K3Surface(D,W) (*function*), 370
K3Surface(g,B) (*function*), 363
K3Surface(x) (*function*), 373, 374
K3SurfaceRaw(D,i) (*function*), 374
K3SurfaceRaw(D,Q,i) (*function*), 374
K3SurfaceToRecord(X) (*function*), 373
Kawamata Blowup (*example*), 400
Kernel(f) (*function*), 124
Kodaira-Enriques Types of Three Surfaces (*example*), 244
KodairaConfigurations(imat) (*function*), 322
KodairaDimension(S) (*function*), 244
KodairaEnriquesType(S) (*function*), 243
KummerSurfaceScheme(C) (*function*), 237

L

L eq K (*operation*), 96
L meet K (*operation*), 99
L-Polynomial from Point Counts (*example*), 187
Large Integral Points on Elliptic Curves (*example*), 108

LatticeElementToMonomial(D,v) (*function*), 403
LCM(D1, D2) (*function*), 200
Lcm(D1, D2) (*function*), 200
LCT(C) (*function*), 161
LCT(C, P) (*function*), 161
LeastCommonMultiple(D1, D2) (*function*), 200
Length(C) (*function*), 396
Length(X) (*function*), 13
Lengths(X) (*function*), 14
Lift(a, P) (*function*), 195
Lift(i, P) (*function*), 195
Lifting a p-adic Point (*example*), 53
LiftPoint(F, d, P, n) (*function*), 53
LiftPoint(P, n) (*function*), 53
Line(C,p,q) (*function*), 149
Line(P,S) (*function*), 149
Linear System of Plane Conics (*example*), 7
Linear Systems from Subspaces (*example*), 98
LinearCovariants(f) (*function*), 314
LinearElimination(S) (*function*), 107
LinearlyEquivalentDivisorWithNoSupportOn(D,S) (*function*), 403
LinearSystem(L, P) (*function*), 87
LinearSystem(L, p) (*function*), 87
LinearSystem(L, P, M) (*function*), 89
LinearSystem(L, p, m) (*function*), 89
LinearSystem(L, P, M, T) (*function*), 92
LinearSystem(L, p, m, t) (*function*), 92
LinearSystem(L, X : parameters) (*function*), 89
LinearSystem(L,F) (*function*), 96
LinearSystem(L,V) (*function*), 96
LinearSystem(P, d) (*function*), 86
LinearSystem(P, F) (*function*), 86
LinearSystemDivisorRestriction(S,B,ms) (*function*), 259
LinearSystemTrace(L, X) (*function*), 95
LineAtInfinity(A) (*function*), 167
Linking(u,v) (*function*), 230
LinkingNumbers(s) (*function*), 230
Local Intersection of Tangent Curves (*example*), 164
LocalBlowUp(X,Y) (*function*), 18
LocalIntersectionNumber(e) (*function*), 270
Log Canonical Threshold over an Extension (*example*), 162
Log Canonical Thresholds of Plane Cubics (*example*), 161
LogCanonicalThreshold(C) (*function*), 161
LogCanonicalThreshold(C, P) (*function*), 161
LogCanonicalThresholdAtOrigin(C) (*function*), 161
LogCanonicalThresholdOverExtension(C) (*function*), 161
LPolynomial(C) (*function*), 185
LPolynomial(C, m) (*function*), 185
LPolynomialFromNumbersOfPoints(n, q) (*function*), 187
LPolynomialFromTracesOfFrobenius(t, q) (*function*), 187

M

MagicNumber(C) (*function*), 359
MakeBasket(Q) (*function*), 359
MakePCMap(A, P, S) (*procedure*), 70
MakePCMap(m) (*procedure*), 70
MakeProjectiveClosureMap(A, P, S) (*procedure*), 70
MakeProjectiveClosureMap(m) (*procedure*), 70
MakeResolutionGraph(g,s) (*function*), 227
MakeResolutionGraph(g,s,t) (*function*), 227
MakeResolutionGraph(N) (*function*), 227
MakeSpliceDiagram(e,l,a) (*function*), 229
MakeSpliceDiagram(g,e,a) (*function*), 229
Map from a Hirzebruch Surface (*example*), 406
Map from the Line to the Plane (*example*), 6
map< X -> Y | F > (*constructor*), 56
map< X -> Y | F, G > (*constructor*), 56
map< X -> Y | Q > (*constructor*), 58
map< X -> Y | u, F > (*constructor*), 56
Maps Defined by Function Field Elements (*example*), 58
Matrix(f) (*function*), 75
MatrixRepresentation(A) (*function*), 174
MaxCones(F) (*function*), 387
MilnorNumberAnalyticHypersurface(dat) (*function*), 33
Minimal Model of a Blown-Up Enriques Surface (*example*), 248
Minimal Model of a Blown-Up Quintic Surface (*example*), 251
Minimal Model of a Blown-Up Torus (*example*), 247
Minimal Model of a Horikawa Surface (*example*), 250
Minimal Model of a Rational Surface (*example*), 245
Minimal Model Program on a Blowup (*example*), 409
MinimalBasis(X) (*function*), 22
MinimalChernNumber(S,n) (*function*), 240
MinimalDualResolutionGraph(dsd) (*function*), 266
MinimalModelGeneralType(S) (*function*), 249
MinimalModelKodairaDimensionOne(S) (*function*), 248
MinimalModelKodairaDimensionZero(S) (*function*), 246
MinimalModelRationalSurface(S) (*function*), 245
MinimalModelRuledSurface(S) (*function*), 246
MinimiseWeights(~X) (*procedure*), 362
Minimising and Reducing a Plane Quartic (*example*), 213
MinimizeCubicSurface(f, p) (*function*), 309
MinimizeDeg4delPezzo(f, p) (*function*), 310

MinimizeReduce(C) (*function*), 212
MinimizeReduce(S) (*function*), 310
MinimizeReduceCubicSurface(f) (*function*), 309
MinimizeReduceDeg4delPezzo(f) (*function*), 310
MinimizeTernaryForm(F) (*function*), 212
MinimizeTernaryFormAtp(F,p) (*function*), 212
Minimizing Cubic and Quartic Del Pezzo Surfaces (*example*), 310
MinRedTernaryForm(F) (*function*), 212
MMP(X) (*function*), 409
ModifySelfintersection(~v,n) (*procedure*), 228
ModifyTransverseIntersection(~v,n) (*procedure*), 228
Module(L) (*function*), 190
Module(S) (*function*), 121, 183
ModuleHomomorphism(f) (*function*), 124
MonodromyRepresentation(X) (*function*), 335
MonomialLattice(C) (*function*), 398
MonomialLattice(X) (*function*), 393
MonomialsOfWeightedDegree(X, D) (*function*), 86
MoriCone(X) (*function*), 407
Movable Divisors on a Blowup (*example*), 402
MovablePart(D) (*function*), 401
Multidegree(X,f) (*function*), 14
Multiplicities(D,P) (*function*), 102
Multiplicities(g) (*function*), 228
Multiplicities(M) (*function*), 211
Multiplicities(S,D) (*function*), 258
Multiplicities(S,dsd,D) (*function*), 258
MultiplicitiesAndIntersections(S,D) (*function*), 258
MultiplicitiesAndIntersections(S,dsd,D) (*function*), 258
Multiplicity(C, p) (*function*), 158
Multiplicity(D,E) (*function*), 102
Multiplicity(L,p) (*function*), 98
Multiplicity(p) (*function*), 32, 158
Multiplicity(X,p) (*function*), 32
MultiplicityFast(D,E) (*function*), 102
MultiplyDivisor(n, D , D0) (*function*), 206

N

n * D (*operation*), 104, 198, 400
NagataAutomorphism(A) (*function*), 74
Name(A,i) (*function*), 10
Name(C,i) (*function*), 397
Name(X,i) (*function*), 20
Naming Affine Coordinates (*example*), 10
NefCone(X) (*function*), 407
NegativePrimeDivisors(D) (*function*), 106
Next(P) (*function*), 10
Ngens(A) (*function*), 174
NGrad(X) (*function*), 13
NoCommonComponent(X,Y) (*function*), 22
NoetherNormalisation(X) (*function*), 361
NoetherNumerator(X) (*function*), 361
NoetherWeights(X) (*function*), 361
NonOrdinaryPrimes(f6,lim) (*function*), 366
NonSimplicialCones(F) (*function*), 387
Normal Forms of Isolated Singularities (*example*), 39
NormalFan(F,C) (*function*), 383
NormalFan(P) (*function*), 383
NormalFormOfHypersurfaceSingularity(f) (*function*), 36
Normalisation(phi) (*function*), 62
Normalising a Coordinate Ring (*example*), 139
Normalization(phi) (*function*), 62
NormalNumber(C) (*function*), 358
Number and Index of a K3 Surface (*example*), 370
Number(D,X) (*function*), 369
NumberOfAffinePatches(X) (*function*), 47
NumberOfBlowUpDivisors(dsd) (*function*), 256
NumberOfCoordinates(X) (*function*), 13
NumberOfGenerators(A) (*function*), 174
NumberOfGradings(C) (*function*), 397
NumberOfGradings(X) (*function*), 13, 394
NumberOfPlacesDegECF(C, m) (*function*), 184
NumberOfPlacesOfDegreeOneECF(C) (*function*), 184
NumberOfPlacesOfDegreeOneECF(C, m) (*function*), 184
NumberOfPlacesOfDegreeOneECFBound(C) (*function*), 184
NumberOfPlacesOfDegreeOneECFBound(C, m) (*function*), 184
NumberOfPlacesOfDegreeOneOverExactConstantField(C) (*function*), 184
NumberOfPlacesOfDegreeOneOverExactConstantField(C, m) (*function*), 184
NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(C) (*function*), 184
NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(C, m) (*function*), 184
NumberOfPlacesOfDegreeOverExactConstantField(C, m) (*function*), 184
NumberOfPointsOnCubicSurface(f) (*function*), 311
NumberOfPointsOnMinimalResolutionFibre(dsd) (*function*), 270
NumberOfPointsOnResolutionFibre(dsd) (*function*), 270
NumberOfPunctures(C) (*function*), 165
NumberOfQuotientGradings(C) (*function*), 397
NumberOfQuotientGradings(X) (*function*), 394
NumbersOfPointsFromLPolynomial(L) (*function*), 187
NumbersOfPointsOnDegree2K3Surface(f6,p,d) (*function*), 366
Numerator(D) (*function*), 199
Numerator(f) (*function*), 24
Numerator(f, X) (*function*), 25
Numerator(X) (*function*), 361
NumericClebschTransfer(f, inv, p)

(*function*), 314

O

OneParameterSubgroupsLattice(C) (*function*), 398
OneParameterSubgroupsLattice(X) (*function*), 393
OneSkeleton(F) (*function*), 386
Order(A) (*function*), 174
Order(f) (*function*), 173
Origin(A) (*function*), 15, 148
OuterNormals(F) (*function*), 388

P

p @ f (*operation*), 15, 182
p @@ f (*operation*), 66
p div k (*operation*), 194
P eq Q (*operation*), 26, 192, 194
p eq q (*operation*), 27, 157, 357
p in C (*operation*), 158
P in S (*operation*), 194
p in X (*operation*), 27
p mod k (*operation*), 194
P ne Q (*operation*), 192, 194
P notin S (*operation*), 194
p1 + p2 (*operation*), 194
p1 - p2 (*operation*), 194
Parametrization(C, p) (*function*), 195
Parametrization(C, p, P) (*function*), 195
ParametrizeDegree5DelPezzo(X) (*function*), 308
ParametrizeDegree6DelPezzo(X) (*function*), 305
ParametrizeDegree7DelPezzo(X) (*function*), 305
ParametrizeDegree8DelPezzo(X) (*function*), 303
ParametrizeDegree9DelPezzo(X) (*function*), 303
ParametrizeDelPezzo(X, P2) (*function*), 300
ParametrizeDelPezzoDeg6(X) (*function*), 306
ParametrizePencil(phi, P2) (*function*), 300
ParametrizeProjectiveHypersurface(X, P2) (*function*), 295
ParametrizeProjectiveSurface(X, P2) (*function*), 296
ParametrizeQuadric(X,P2) (*function*), 299
ParametrizeSingularDegree3DelPezzo(X,P2) (*function*), 308
ParametrizeSingularDegree4DelPezzo(X,P2) (*function*), 308
Parametrizing a Degree 6 Del Pezzo Surface (*example*), 306
Parametrizing a Ruled Surface (*example*), 300
Parametrizing a Singular Cubic Del Pezzo (*example*), 308
Parametrizing an Anticanonical Sphere (*example*), 304
Parametrizing Factors of an Affine Hypersurface (*example*), 299
Parametrizing Quadric Surfaces (*example*), 299
Parametrizing Rational Hypersurfaces (*example*), 296
Parent(D) (*function*), 401
Patches of Weighted Projective Space (*example*), 48
PCMap(A) (*function*), 50
Pencil of Curves with Tangency Conditions (*example*), 94
Pentahedron of a Random Cubic Surface (*example*), 315
PentahedronIdeal(f) (*function*), 315
Period Matrices of a Hyperelliptic Curve (*example*), 347
Permutation Automorphisms of Affine Space (*example*), 73
PermutationAutomorphism(A, g) (*function*), 73
PermutationGroup(A) (*function*), 174
PermutationRepresentation(A) (*function*), 174
phi in A (*operation*), 174
PicardClass(D) (*function*), 401
PicardLattice(X) (*function*), 393
PicardToClassGroupsMap(X) (*function*), 403
PicardToClassLatticesMap(X) (*function*), 403
Place(C, I) (*function*), 192
Place(p) (*function*), 192
Place(Q) (*function*), 192
Places from Rational Functions (*example*), 193
Places(C) (*function*), 192
Places(C, m) (*function*), 192
Places(p) (*function*), 192
Plane Model of X0(58) (*example*), 219
PluriCanonicalBasis(S,m) (*function*), 261
Plurigenus(S,n) (*function*), 239
PlurigenusOfDesingularization(S,m) (*function*), 291
Point Access Functions on a Riemann Surface (*example*), 341
Point(r,n,Q) (*function*), 357
Point(r,Q) (*function*), 357
Point(X, S) (*function*), 340
PointOnRegularModel(M, x) (*function*), 212
Points of a Cluster over a Number Field (*example*), 30
Points of an Affine Space (*example*), 148
Points on a Scheme over a Finite Field (*example*), 28
Points(B) (*function*), 359
Points(X) (*function*), 27
Points(X,L) (*function*), 27
PointsAtInfinity(C) (*function*), 167
PointsCubicModel(C, B : parameters) (*function*), 210
PointSearch(S,H : parameters) (*function*), 54
PointSet(X,L) (*function*), 26
PointSet(X,m) (*function*), 26
PointsInGeneralPosition(P2,S) (*function*), 302
PointsModPN(S, M) (*function*), 28

PointsModPN(S, P, N) (*function*), 28
PointsOverAlgebraicClosure(Z) (*function*), 30
PointsOverDiscriminantPoint(X, k) (*function*), 341
PointsOverSplittingField(Z) (*function*), 30
Polarisation(p) (*function*), 357
PolarisedVariety(d,W,n) (*function*), 360
Poles(C, f) (*function*), 194
Poles(f) (*function*), 193
Polyhedron(D) (*function*), 403
Polynomial with Mathieu Monodromy (*example*), 112
Polynomial(X) (*function*), 22
PolynomialMap(L) (*function*), 98
Polynomials(X) (*function*), 22
PossibleCanonicalDissidentPoints(C) (*function*), 360
PossibleSimpleCanonicalDissidentPoints(C) (*function*), 360
Precision(X) (*function*), 334
PrimaryComponents(X) (*function*), 43
Prime Components of a Scheme (*example*), 44
PrimeComponents(X) (*function*), 43
PrimeFactorisation(D) (*function*), 102
PrincipalDivisor(C, f) (*function*), 197
PrincipalDivisor(D, f) (*function*), 197
PrincipalDivisor(f) (*function*), 197
ProductProjectiveSpace(k,N) (*function*), 12
Proj(D) (*function*), 402
Proj(R) (*function*), 10, 17
Projection(X) (*function*), 58
Projection(X, p) (*function*), 58
Projection(X, Q) (*function*), 58
Projection(X,Y) (*function*), 58
ProjectionFromNonsingularPoint(X,p) (*function*), 59
Projective Map on a Plane Quartic (*example*), 59
Projective Plane from Its Fan (*example*), 376
ProjectiveClosure(A) (*function*), 166
ProjectiveClosure(C) (*function*), 167
ProjectiveClosure(f) (*function*), 70
ProjectiveClosure(X) (*function*), 46
ProjectiveClosureMap(A) (*function*), 50
ProjectiveFunction(f) (*function*), 24, 182
ProjectiveLine(k) (*function*), 9
ProjectiveLineProcess(k, n) (*function*), 10
ProjectiveLineProcess(V) (*function*), 10
ProjectiveMap(f) (*function*), 59
ProjectiveMap(f, Y) (*function*), 59
ProjectiveMap(L) (*function*), 59
ProjectiveMap(L, Y) (*function*), 59
ProjectivePlane(k) (*function*), 147
ProjectiveRationalFunction(f) (*function*), 24
ProjectiveSpace(k,n) (*function*), 9, 147, 392
ProjectiveSpace(k,W) (*function*), 9, 392
ProjectiveSpace(R) (*function*), 10
ProjectiveSpaceAsToricVariety(k,n) (*function*), 392
ProjectiveSpaceAsToricVariety(k,W) (*function*), 392
Projectivities of Affine Space (*example*), 74
Projectivity(A,M) (*function*), 74
Prune(phi) (*function*), 62
Pullback(f, p) (*function*), 66
Pullback(f, X) (*function*), 67
Pullback(f,L) (*function*), 99
Pullback(phi, X) (*function*), 170
PureRayIndices(F) (*function*), 388
PureRays(F) (*function*), 388
Pushforward and Pullback by a Map (*example*), 171
Pushforward(phi, X) (*function*), 171
p[i] (*literal*), 15, 148, 157

Q

Q in X (*operation*), 27
QFactorialisation(X) (*function*), 407
QuadraticTransformation(P) (*function*), 77
QuadraticTransformation(P,Q) (*function*), 77
QuadraticTransformation(X) (*function*), 77
QuadraticTransformation(X,Q) (*function*), 77
Quadrifolium (*example*), 92
Quartic Curve with a Tacnode (*example*), 92
Quotient of the Klein Quartic (*example*), 180
QuotientGradings(C) (*function*), 397
QuotientGradings(X) (*function*), 394
Quotrem(D, n) (*function*), 198
Quotrem(p1, k) (*function*), 194

R

r * D (*operation*), 104
RamificationDivisor(C) (*function*), 198
RamificationDivisor(D) (*function*), 203
RamificationDivisor(m) (*function*), 170
RamificationIndex(P) (*function*), 341
RamificationPoints(X) (*function*), 335
Random Curves (*example*), 153
Random(LS) (*function*), 96
Random(S) (*function*), 28
RandomAbelianSurface_d10g6(P) (*function*), 253
RandomCompleteIntersection(P,ds) (*function*), 237
RandomCurveByGenus(g, K) (*function*), 153
RandomDivisor(X,d) (*function*), 343
RandomEllipticFibration_d10g10(P) (*function*), 253
RandomEllipticFibration_d7g6(P) (*function*), 253
RandomEllipticFibration_d8g7(P) (*function*), 253
RandomEllipticFibration_d9g7(P) (*function*), 253
RandomEnriquesSurface_d9g6(P) (*function*), 252
RandomNodalCurve(d, g, P) (*function*), 153

RandomOrdinaryPlaneCurve(d, S, P) (*function*), 153
RandomPlace(C, m) (*function*), 192
RandomPoint(X) (*function*), 341
RandomRationalSurface_d10g9(P) (*function*), 252
Rational Points on a Plane Cubic (*example*), 211
Rational Quartic Curve as an Image (*example*), 67
Rational Surface of Degree 10 (*example*), 130
RationalPoints(X) (*function*), 27
RationalPoints(X,L) (*function*), 27
RationalPoints(Z) (*function*), 30
RationalPoints(Z,L) (*function*), 30
RationalPointsByFibration(X) (*function*), 28
RationalRuledSurface(P,n) (*function*), 237
RationalScroll(k,s,A) (*function*), 393
RawBasket(X) (*function*), 360
Ray(F,i) (*function*), 388
RayLattice(C) (*function*), 398
RayLatticeMap(C) (*function*), 398
Rays(F) (*function*), 388
Rays(X) (*function*), 393
Reconstructing a Divisor from Its Data (*example*), 196
ReduceCluster(C) (*function*), 213
ReduceCluster(X) (*function*), 213
ReduceCubicSurface(f) (*function*), 309
ReducedFactorisation(D) (*function*), 102
ReducedSubscheme(X) (*function*), 43
ReducePlaneCurve(C) (*function*), 212
ReducePlaneCurve(f) (*function*), 212
ReduceTernaryForm(f) (*function*), 212
Reducibility from Too Many Conditions (*example*), 97
Reducing a Scheme Modulo p (*example*), 55
Reduction(D) (*function*), 202
Reduction(D, A) (*function*), 202
Reduction(D,p) (*function*), 106
Reduction(I, p) (*function*), 55
Reduction(L) (*function*), 96
Reduction(X, p) (*function*), 54
RegularModel(C, P) (*function*), 211
RegularSpliceDiagram(P) (*function*), 229
Relations(L) (*function*), 190
Relations(L, m) (*function*), 190
Relations(S) (*function*), 183
Relations(S, m) (*function*), 183
Relative Minimal and Canonical Models (*example*), 411
RelativeProj(D) (*function*), 402
RemoveLinearRelations(X) (*function*), 18
RemoveWeight(~X,w) (*procedure*), 362
RemoveWeight(X,w) (*function*), 363
Repeated Patches and Closures (*example*), 47
Representation(P) (*function*), 340
Representative(X,m) (*function*), 399, 400
RepresentativePoint(P) (*function*), 194
Residue(a, P) (*function*), 195
Residue(d, P) (*function*), 190
ResidueClassField(P) (*function*), 195
Resolution Graph of a Curve Singularity (*example*), 225
Resolution Graph of a Pencil (*example*), 226
Resolution Graph with Three Meeting Divisors (*example*), 269
Resolution(F) (*function*), 390
Resolution(X) (*function*), 407
ResolutionGraph(C, p) (*function*), 224
ResolutionGraph(P) (*function*), 225
ResolutionGraph(P,a,b) (*function*), 226
ResolutionGraph(v) (*function*), 224
ResolutionGraphVertex(g,i) (*function*), 223
ResolveAffineCurve(p) (*function*), 284
ResolveAffineMonicSurface(s) (*function*), 287
ResolveFanMap(F1,F2) (*function*), 389
ResolveLinearSystem(D) (*function*), 407
ResolveProjectiveCurve(p) (*function*), 285
ResolveSingByBlowUp(X,Y) (*function*), 255
ResolveSingularSurface(S) (*function*), 254
Restricting Maps to Patches (*example*), 70
Restriction(f, Y) (*function*), 25
Restriction(f,X,Y) (*function*), 61
Restriction(S, Y) (*function*), 123
RestrictionOfScalars(S, F) (*function*), 50
RestrictionToPatch(f, Xi) (*function*), 24
RestrictionToPatch(f,i,j) (*function*), 70
RestrictionToPatch(f,j) (*function*), 70
RestrictionToSubtorus(Z) (*function*), 393
Riemann Surface over a Number Field (*example*), 331
Riemann Surface over the Rationals (*example*), 330
Riemann-Roch Space by Cech Cohomology (*example*), 404
Riemann-Roch Space on a Blown-Up Plane (*example*), 403
RiemannRochBasis(D) (*function*), 106, 403
RiemannRochBasis(X, I) (*function*), 125
RiemannRochCoordinates(f,D) (*function*), 107
RiemannRochDimension(D) (*function*), 403
RiemannRochPolytope(D) (*function*), 403
RiemannRochSpace(D) (*function*), 106, 203
RiemannSurface(D) (*function*), 343
RiemannSurface(f) (*function*), 330
RiemannSurface(f,sigma) (*function*), 330
RiemannSurface(L,m) (*function*), 332
RiemannSurface(P) (*function*), 340
RiemannSurface(p,m) (*function*), 332
Ring(P) (*function*), 27
RingMap(P) (*function*), 27
RootVertex(s) (*function*), 229
RoundDownDivisor(D) (*function*), 101
RoundUpDivisor(D) (*function*), 101
RuledSurface(k,a,b) (*function*), 12, 147

RuledSurface(k,a1,a2) (*function*), 393
RuledSurface(k,n) (*function*), 12, 147, 393
Ruling on a Quadric Cone (*example*), 123

S

S eq T (*operation*), 189
s eq t (*operation*), 230
S in C (*operation*), 158
S subset X (*operation*), 27
Salmon's First Invariant (*example*), 315
Saturate(~X) (*procedure*), 20
SaturateSheaf(~S) (*procedure*), 121
Scheme(F) (*function*), 24
Scheme(P) (*function*), 26
Scheme(p) (*function*), 27
Scheme(S) (*function*), 121
Scheme(X,F) (*function*), 16
Scheme(X,f) (*function*), 16, 413
Scheme(X,I) (*function*), 16
Scheme(X,Q) (*function*), 16, 413
SchemeGraphMap(X, Y, I) (*function*), 79
SchemeGraphMapToSchemeMap(f) (*function*), 80
SchemeMap(f) (*function*), 173
Searching for Points on a Surface (*example*), 54
Secant Variety (*example*), 83
SecantVariety(X) (*function*), 82
Second Affine Patch (*example*), 167
Sections(L) (*function*), 96
Segre Embedding of a Product (*example*), 12
SegreEmbedding(X) (*function*), 12
SegreProduct(Xs) (*function*), 12
Self-Intersection in a Hirzebruch Surface (*example*), 379
SelfIntersection(D) (*function*), 105
SelfIntersection(dsd,i) (*function*), 270
SelfIntersection(G,i) (*function*), 270
SelfIntersectionOfStrictTransform(S,D) (*function*), 259
SelfIntersections(g) (*function*), 228
SerreBound(C) (*function*), 184
SerreBound(C, m) (*function*), 184
SetVerbose("Degree2K3", v) (*procedure*), 366
SetVerbose("ParamDP", v) (*procedure*), 303
SetVerbose("WeilPolynomials", v) (*procedure*), 364
Sheaf of an Unsaturated Module (*example*), 121
Sheaf(D) (*function*), 106
Sheaf(M, X) (*function*), 119
SheafHomomorphism(S, T, h) (*function*), 124
SheafHoms(S, T) (*function*), 123
SheafOfDifferentials(X) (*function*), 120
SheafToDivisor(S) (*function*), 101
Sheaves on a Cubic Surface (*example*), 119
ShortBasis(D) (*function*), 203
SignDecomposition(D) (*function*), 102, 199
Simple Singularities of a Degenerate Del Pezzo (*example*), 242
SimpleCanonicalDissidentPoints(C) (*function*), 360
Simplex(A) (*function*), 15
Simplicial Subdivision of a Cone (*example*), 389
SimplicialSubdivision(C) (*function*), 389
SimplicialSubdivision(F) (*function*), 389
Singular Cones of a Weighted Projective Space (*example*), 387
SingularCones(F) (*function*), 387
Singularities in Weighted Projective Space (*example*), 42
SingularPoint(dsd) (*function*), 256
SingularPoints(C) (*function*), 165
SingularPoints(X) (*function*), 335
SingularPointsOverAlgebraicClosure(C) (*function*), 165
SingularPointsOverSplittingField(C) (*function*), 165
SingularRank(X) (*function*), 363
SingularSubscheme(X) (*function*), 42
SixTangentConicCheck(F,C) (*function*), 318
SixTangentConicCheck(S,C) (*function*), 318
SixTangentConicsModp(F) (*function*), 318
Size(g) (*function*), 228
Size(s) (*function*), 230
Skeleton(F,n) (*function*), 386
SkewInvariant100(f) (*function*), 313
SmallPeriodMatrix(X) (*function*), 334
Solutions to Hall's Conjecture (*example*), 109
Solve(p, F) (*function*), 298
SpaceOfDifferentialsFirstKind(C) (*function*), 189
SpaceOfHolomorphicDifferentials(C) (*function*), 189
Spanning Fan of a Polytope (*example*), 384
SpanningFan(P) (*function*), 383
Spec(R) (*function*), 10, 17
SpliceDiagram(C,p) (*function*), 229
SpliceDiagram(g) (*function*), 231
SpliceDiagram(g,v) (*function*), 231
SpliceDiagram(v) (*function*), 229
SpliceDiagramVertex(s,i) (*function*), 229
StructureSheaf(X) (*function*), 119
StructureSheaf(X, n) (*function*), 119
SubcanonicalCurve(g,d,Q) (*function*), 362
Subsystem from an Incomplete Linear System (*example*), 90
Subsystem on a Product of Weighted Spaces (*example*), 90
Subsystems Imposed at Points (*example*), 87
Subsystems through Thousands of Points (*example*), 89
Superelliptic Riemann Surface of Genus 12 (*example*), 332
SuperScheme(X) (*function*), 21
Support(D) (*function*), 102, 197, 343

Surface and 3-Fold Point Singularities (*example*), 356
Surface(A,f) (*function*), 236
Surface(A,I) (*function*), 236
Surface(A,S) (*function*), 236

T

Tangent Variety (*example*), 82
TangentCone(C, p) (*function*), 158
TangentCone(p) (*function*), 32, 158
TangentCone(X,p) (*function*), 32
TangentLine(C, p) (*function*), 158
TangentLine(p) (*function*), 158
TangentSheaf(X) (*function*), 120
TangentSpace(p) (*function*), 32
TangentSpace(X,p) (*function*), 32
TangentVariety(X) (*function*), 81
TensorPower(S, n) (*function*), 122
TensorProduct(S, T) (*function*), 122
TerminalIndex(p) (*function*), 357
Terminalisation(F) (*function*), 390
Terminalisation(X) (*function*), 407
TerminalPolarisation(p) (*function*), 357
Testing for Hyperellipticity (*example*), 168
Testing Local Solvability (*example*), 53
Testing Scheme Inclusion and Equality (*example*), 23
Testing the Jacobian Conjecture (*example*), 72
TjurinaNumberAnalyticHypersurface(dat) (*function*), 33
Toric Variety from a Cox Ring (*example*), 397
Toric Variety with a Fan Like the Plane (*example*), 395
ToricAffinePatch(X,i) (*function*), 395
ToricAffinePatch(X,S) (*function*), 395
ToricIdentityMap(X) (*function*), 406
ToricVariety(C) (*function*), 397
ToricVariety(G) (*function*), 399
ToricVariety(k) (*function*), 392
ToricVariety(k,F) (*function*), 393
ToricVariety(k,M,v) (*function*), 392
ToricVariety(k,n) (*function*), 392
ToricVariety(k,Z) (*function*), 392
ToricVariety(k,Z,Q) (*function*), 392
ToricVarietyMap(X,Y) (*function*), 406
ToricVarietyMap(X,Y,f) (*function*), 406
TotalLinking(v) (*function*), 230
Trace on a Twisted Cubic (*example*), 95
TracesOfFrobeniusFromLPolynomial(L) (*function*), 187
Translation to Infinity (*example*), 169
Translation(A,p) (*function*), 73, 169
Translation(P,p,q) (*function*), 76
Translation(P,Q) (*function*), 76
Translation(X,p) (*function*), 76
TranslationOfSimplex(P,Q) (*function*), 76
Translations of Schemes (*example*), 76
TranslationToInfinity(C,p) (*function*), 169
TransverseIndex(C) (*function*), 358
TransverseIntersections(g) (*function*), 228
TransverseType(C) (*function*), 358
Trigonal Curve (*example*), 207
Trigonal Curve in a Toric Variety (*example*), 414
Trigonal Map of a Canonical Curve (*example*), 214
TriTangentLines(F) (*function*), 318
TriTangentLines(S) (*function*), 318
Twist(S, n) (*function*), 119
Twisted Cubics (*example*), 113
Two Cubics Meeting with Multiplicity 8 (*example*), 94
TwoGenerators(P) (*function*), 193
TwoGenus(X) (*function*), 363
TypeOfContraction(X,i) (*function*), 407
TypesOfContractions(X) (*function*), 407

U

UnderlyingGraph(g) (*function*), 227
UnderlyingGraph(s) (*function*), 229
UnderlyingRing(C) (*function*), 396
UnderlyingVertex(v) (*function*), 229
UniformizingParameter(P) (*function*), 195
UniformizingParameter(p) (*function*), 183
Union(C,D) (*function*), 150
Union(X,Y) (*function*), 17
Unique Quadric Containing a Curve (*example*), 90
Uniqueness of Projective Closure (*example*), 167

V

v eq w (*operation*), 224, 230
Valency(v) (*function*), 230
Valuation(a, P) (*function*), 195
Valuation(d, P) (*function*), 190
Valuation(D,P) (*function*), 200
Valuation(D,p) (*function*), 200
Valuation(f, P) (*function*), 194
Valuation(f, p) (*function*), 183
Valuation(P) (*function*), 194
Valuation(p) (*function*), 183
Variety(D) (*function*), 401
Variety(G) (*function*), 100
Verbose Output for Riemann Surfaces (*example*), 329
Vertex(v) (*function*), 224, 229
VertexLabels(s) (*function*), 230
VertexPath(u,v) (*function*), 230
Vertices(s) (*function*), 229
VirtualRayIndices(F) (*function*), 388
VirtualRays(F) (*function*), 388

W

Weierstrass Model of an Elliptic Fibration

(*example*), 326
WeierstrassPlaces(C) (*function*), 192
WeierstrassPlaces(D) (*function*), 203
WeierstrassPoints(C) (*function*), 192
WeierstrassPoints(D) (*function*), 203
Weighted Blowup (*example*), 160
WeightedAffinePatch(P, i) (*function*), 47
WeightedProjectiveSpace(k,n) (*function*), 392
WeightedProjectiveSpace(k,W) (*function*), 392
Weights of a Flip (*example*), 408
Weights(X) (*function*), 360
WeightsOfFlip(X,i) (*function*), 408
Weil Polynomial of a Degree 2 K3 Surface (*example*), 366
Weil Polynomial of a Quartic Surface (*example*), 365
Weil(D) (*function*), 401
WeilPolynomialOfDegree2K3Surface(f6) (*function*), 366
WeilPolynomialOverFieldExtension(f, deg) (*function*), 364
WeilPolynomialToRankBound(f, q) (*function*), 364
WeilRestriction(S, F) (*function*), 50
WeilToClassGroupsMap(C) (*function*), 398
WeilToClassLatticesMap(C) (*function*), 398
WPS(k,n) (*function*), 392
WPS(k,W) (*function*), 392
WriteK3Data(Q,F) (*procedure*), 373
WronskianOrders(C) (*function*), 184
WronskianOrders(D) (*function*), 203

X

X ! Q (*operation*), 27
x * f (*operation*), 189
x + y (*operation*), 189
X . i (*operation*), 20
x / r (*operation*), 189
x / y (*operation*), 189
X @ f (*operation*), 174
X @@ f (*operation*), 67, 175
x - y (*operation*), 189
X eq Y (*operation*), 23, 361
X in L (*operation*), 99
X join Y (*operation*), 17
X meet Y (*operation*), 17
X(L) ! Q (*operation*), 27
X(L) (*function*), 26
X(m) (*function*), 26

Z

ZariskiDecomposition(D) (*function*), 106
ZeroDivisor(X) (*function*), 101, 343, 399
ZeroFan(L) (*function*), 383
Zeros and Poles (*example*), 194
Zeros(C, f) (*function*), 194
Zeros(f) (*function*), 193
ZeroSubscheme(S, s) (*function*), 130
Zeta Function of an Affine Weierstrass Model (*example*), 186
Zeta Functions of Resolution Fibres (*example*), 271
ZetaFunction(C) (*function*), 185
ZetaFunction(C, m) (*function*), 185
ZetaFunction(f, p) (*function*), 186
ZetaFunctionOfCurveModel(C) (*function*), 186
ZetaFunctionOfMinimalResolutionFibre(dsd) (*function*), 271
ZetaFunctionOfResolutionFibre(dsd) (*function*), 270, 271